Roberto Giacobazzi
Josh Berdine
Isabella Mastroeni (Eds.)

LNCS 7737
ARCoSS

Verification, Model Checking, and Abstract Interpretation

14th International Conference, VMCAI 2013
Rome, Italy, January 2013
Proceedings

Springer

Lecture Notes in Computer Science 7737

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

Advanced Research in Computing and Software Science
Subline of Lectures Notes in Computer Science

Roberto Giacobazzi Josh Berdine
Isabella Mastroeni (Eds.)

Verification, Model Checking, and Abstract Interpretation

14th International Conference, VMCAI 2013
Rome, Italy, January 20-22, 2013
Proceedings

Springer

Volume Editors

Roberto Giacobazzi
University of Verona, Department of Computer Science
Strada Le Grazie 15, 37134 Verona, Italy
E-mail: roberto.giacobazzi@univr.it

Josh Berdine
Microsoft Research
7 JJ Thomson Avenue, Cambridge, CB3 0FB, UK
E-mail: jjb@microsoft.com

Isabella Mastroeni
University of Verona, Department of Computer Science
Strada Le Grazie 15, 37134 Verona, Italy
E-mail: isabella.mastroeni@univr.it

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35872-2 e-ISBN 978-3-642-35873-9
DOI 10.1007/978-3-642-35873-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955045

CR Subject Classification (1998): F.3.1-2, D.2.4, C.2.4, F.4.1, F.1.1, I.2.2

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

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

Printed on acid-free paper

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

Preface

This volume contains the papers presented at VMCAI 2013, the 14th International Conference on Verification, Model Checking, and Abstract Interpretation, held during January 20–22, 2012, in Rome, co-located with POPL 2013 (the 40th ACM SIGPLAN/SIGACT Symposium on Principles of Programming languages). Previous meetings were held in Port Jefferson (1997), Pisa (1998), Venice (2002), New York (2003), Venice (2004), Paris (2005), Charleston (2006), Nice (2007), San Francisco (2008), Savannah (2009), Madrid (2010), Austin (2011), and Philadelphia (2012).

VMCAI is a major conference dealing with state-of-the art research in analysis and verification of programs and systems, with particular emphasis on the cross-fertilization among communities that have developed different methods and models for code and system verification and analysis. VMCAI topics include: program verification, model checking, abstract interpretation and abstract domains, program synthesis, static analysis, type systems, deductive methods, program certification, debugging techniques, program transformation, optimization, hybrid and cyber-physical systems.

This year we had 72 submissions. Each submission was reviewed by at least three Program Committee members. The committee decided to accept 27 papers, with an acceptance rate of 37%. The principal selection criteria were relevance, quality, and originality. We are glad to include in the proceedings the contributions of four invited keynote speakers: Leonardo de Moura (Microsoft Research, USA) on "A Model-Constructing Satisfiability Calculus," Andreas Podelski (Freiburg, Germany) on "Automata as Proofs," Francesco Ranzato (University of Padova, Italy) on "Complete Abstractions Everywhere," and Eran Yahav (Technion, Israel) on "Abstraction-Guided Synthesis."

We would like to thank the members of the Program Committee and all sub-reviewers for their dedicated effort in evaluating and selecting the papers to be presented at the conference. Our gratitude goes to the Steering Committee members for their helpful advice and support, in particular to Lenore Zuck and Dave Schmidt for their constant assistance and invaluable experience with the organization of VMCAI. We would like to thank Liù Catena for the local arrangements in Rome and for her terrific energy in organizing the logistics of all events co-located with POPL2013. Moreover, we would like to thank Matthew Might for the great help in coordinating the events co-located with POPL2013 and the staff of the Hotel Parco dei Principi for their help in making this event

easier to organize. Finally, we are also grateful to Andrei Voronkov for having set up the EasyChair system that was used to handle the whole scientific production workflow.

November 2012

Roberto Giacobazzi
Josh Berdine
Isabella Mastroeni

Organization

Program Committee

Josh Berdine	Microsoft Research, UK
Nikolaj Bjorner	Microsoft Research, USA
Sandrine Blazy	IRISA - Université Rennes 1, France
Agostino Cortesi	Università Ca' Foscari di Venezia, Italy
Mads Dam	KTH, Sweden
Michael Emmi	Université Paris Diderot (Paris 7), France
Azadeh Farzan	University of Toronto, Canada
Pierre Ganty	IMDEA Software Institute, Spain
Samir Genaim	Universidad Complutense de Madrid, Spain
Roberto Giacobazzi	University of Verona, Italy
Orna Grumberg	Technion - Israel Institute of Technology
Klaus Havelund	Jet Propulsion Laboratory, California Institute of Technology, USA
Jochen Hoenicke	University of Freiburg, Germany
Sebastian Hunt	City University, London, UK
Limin Jia	Carnegie Mellon University, USA
Andy King	University of Kent, UK
Arun Lakhotia	University of Louisiana at Lafayette, USA
Akash Lal	Microsoft Research, India
Rupak Majumdar	UCLA, USA
Matthieu Martel	Université de Perpignan Via Domitia, France
Isabella Mastroeni	University of Verona, Italy
Ganesan Ramalingam	Microsoft Research, India
Roberto Sebastiani	DISI, University of Trento, Italy
Saurabh Srivastava	University of California, Berkeley, USA
Greta Yorsh	IBM Research, USA
Enea Zaffanella	University of Parma, Italy

Additional Reviewers

Adje, Assale	Chapoutot, Alexandre
Amato, Gianluca	Chaudhuri, Swarat
Atig, Mohamed Faouzi	Christ, Juergen
Balliu, Musard	Christ, Jürgen
Barman, Shaon	Costantini, Giulia
Bouissou, Olivier	Dalla Preda, Mila
Bozzelli, Laura	Delahaye, Benoit

Donze, Alexandre
Dragoi, Cezara
Enea, Constantin
Esparza, Javier
Feret, Jerome
Ferrara, Pietro
Gori, Roberta
Guanciale, Roberto
Gupta, Ashutosh
Gurov, Dilian
Hamza, Jad
Hartmanns, Arnd
Hill, Patricia M
Ioualalen, Arnault
Jansen, Nils
Jeannet, Bertrand
Khakpour, Narges
Kincaid, Zachary
Kuperstein, Michael
Lagoon, Vitaly
Legay, Axel
Luccio, Flaminia
Macedonio, Damiano
Madhavan, Ravichandhran
Manevich, Roman
Marin, Andrea
Meller, Yael
Merro, Massimo
Meyerovich, Leo
Murano, Aniello
Parente, Mimmo
Piazza, Carla
Razavi, Niloofar
Rollini, Simone Fulvio
Rozier, Kristin Yvonne
Rybalchenko, Andrey
Samborski-Forlese, Julian
Sangnier, Arnaud
Sankaranarayanan, Sriram
Schuppan, Viktor
Segala, Roberto
Shoham, Sharon
Soffia, Stefano
Sosnovich, Adi
Spalazzi, Luca
Thakur, Aditya
Tonetta, Stefano
Vafeiadis, Viktor
Vasconcelos, Pedro
Vinco, Sara
Vizel, Yakir
Zuliani, Paolo
Zunino, Roberto

Table of Contents

Invited Talks

Session 1: Analysis of Systems with Continuous Behavior

Session 2: Synthesis

Session 3: Analysis Algorithms and Theorem Proving Techniques for Program Analysis

Session 4: Automata-Based Techniques

Session 5: Tools

Session 6: Types and Proof Methodologies

Session 7: Abstract Domains

Session 8: Combining Boolean Solving and Abstract Domains for Theories

Session 9: Distributed/Concurrent System Verification

A Model-Constructing Satisfiability Calculus

Leonardo de Moura[1] and Dejan Jovanović[2]

[1] Microsoft Research
[2] New York University

Abstract. We present a new calculus where recent model-based decision procedures and techniques can be justified and combined with the standard DPLL(T) approach to satisfiability modulo theories. The new calculus generalizes the ideas found in CDCL-style propositional SAT solvers to the first-order setting.

1 Introduction

Considering the theoretical hardness of SAT, the astonishing adeptness of SAT solvers when attacking practical problems has changed the way we perceive the limits of algorithmic reasoning. Modern SAT solvers are based on the idea of *conflict driven clause learning* (CDCL) [11,15,13]. The CDCL algorithm is a combination of an explicit backtracking search for a satisfying assignment complemented with a deduction system based on Boolean resolution. In this combination, the worst-case complexity of both components is circumvented by the components guiding and focusing each other. The generalization of the SAT problem into the first-order domain is called satisfiability modulo theories (SMT). The common way to solve an SMT problem is to employ a SAT solver to enumerate the assignment of the Boolean abstraction of the formula. The candidate Boolean assignment is then either confirmed or refuted by a *decision procedure* dedicated to reasoning about conjunctions of theory-specific constraints. This framework is commonly called DPLL(T) [10,14] and is employed by most of the SMT solvers today. Although DPLL(T) at its core relies on a CDCL SAT solver, this SAT solver is only used as a black-box. This can be seen as an advantage since the advances in SAT easily transfer to performance improvements in SMT. On the other hand, in the last few years the idea of direct model construction complemented with conflict resolution has been successfully generalized to fragments of SMT dealing with theories such as linear real arithmetic [4,12,9], linear integer arithmetic [7], nonlinear arithmetic [8], and floating-point [6]. All these procedures, although quite effective in their corresponding first-order domains, have not seen a more widespread acceptance due to their limitations in purely Boolean reasoning and incompatibility with DPLL(T).

In this paper we propose a *model-constructing satisfiability calculus* (mcSAT) that encompasses all the decision procedures above, including the decision procedures aimed at DPLL(T), while resolving the limitations mentioned above. The mcSAT framework extends DPLL(T) by allowing assignments of variables to concrete values, while relaxing the restriction that decisions, propagations, and explanations of conflicts must be in term of existing atoms.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 1–12, 2013.

2 A Model Based Abstract Procedure

We assume that the reader is familiar with the usual notions and terminology of first-order logic and model theory (for an introduction see e.g. [2]). We describe the new procedure as an abstract transition system in the spirit of Abstract DPLL [14]. The crucial difference of the system we present is that we are not restricted to Boolean decisions. Instead, we allow the model that the theory is trying to construct to be involved in the search and in explaining the conflicts, while allowing new literals to be introduced so as to support more complex conflict analyses.

The states in the transition system are pairs of the form $\langle M, \mathcal{C} \rangle$, where M is a sequence (usually called a *trail*) of trail elements, and $\mathcal{C}$ is a set of clauses. Each trail element is either a *decided literal*, a *propagated literal*, or a *model assignment*. We refer to both decided literals and model assignments as *decisions*. A decided literal is a literal that we assume to be true. On the other hand, a propagated literal, denoted as $C \rightarrow L$, marks a literal L that is implied to be true in the current state by the clause C (the explanation). In both cases, we say that the literal L appears in M, and write this as $L \in M$. A model assignment, written as $x \mapsto \alpha$, is an assignment of a first-order uninterpreted symbol x to a value α.[1] Given a trail M that contains model assignments $x_{i_1} \mapsto \alpha_1, \ldots, x_{i_k} \mapsto \alpha_k$, we can construct a first-order interpretation $\upsilon[M] = [x_{i_1} \mapsto \alpha_1, \ldots, x_{i_k} \mapsto \alpha_k]$. Given a term t, the interpretation $\upsilon[M](t)$ is either a value of the term t under the assignment in M, or undef if the the term cannot be fully evaluated.

The content of the trail implies an interpretation of literals and is the core of our procedure. In order to evaluate the value of some literal L with respect to a trail M, we define the functions $\mathsf{value}_{\mathrm{B}}$ and $\mathsf{value}_{\mathrm{T}}$, the former interpreting the literal according to the Boolean assignment, and the latter interpreting the literal according to the model assignment of variables.

$$\mathsf{value}_{\mathrm{B}}(L, M) = \begin{cases} \mathsf{true} & L \in M \\ \mathsf{false} & \neg L \in M \\ \mathsf{undef} & \text{otherwise} \end{cases} \qquad \mathsf{value}_{\mathrm{T}}(L, M) = \begin{cases} \mathsf{true} & \upsilon[M](L) = \mathsf{true} \\ \mathsf{false} & \upsilon[M](L) = \mathsf{false} \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

We say that a trail M is *consistent* if the Boolean assignment and first-order model are not in conflict, i.e. when for all $L \in M$ we have that $\mathsf{value}_{\mathrm{T}}(L, M) \neq \mathsf{false}$. Additionally we say that the trail M is *complete* when each asserted first-order literal $L \in M$ is justified by the first-order interpretation, i.e. $\mathsf{value}_{\mathrm{T}}(L, M) = \mathsf{true}$. We use the predicate $\mathsf{consistent}(M)$ to denote that M is consistent and $\mathsf{complete}(M)$ to denote that M is complete. Note that if a trail M is consistent, this does not mean that the assertions on the trail are truly satisfiable (feasible), just that the current partial assignment does not refute any of the individual trail literals. When there is a set of literals on the

[1] The actual representation for values is theory specific and depends on the type of x. For example, for the theory of liner real arithmetic, the values are rational numbers. We never assign Boolean variables to values as they are considered literals.

trail M that, together with the model assignments from M, is not satisfiable, we call the trail *infeasible* and denote this with the predicate $\mathsf{infeasible}(M)$. We have that $\neg\,\mathsf{consistent}(M)$ implies $\mathsf{infeasible}(M)$.

Since the values of $\mathsf{value}_{\mathrm{T}}(L, M)$ and $\mathsf{value}_{\mathrm{B}}(L, M)$ do not disagree for all $L \in M$, we define the value of a literal in a consistent state as

$$\mathsf{value}(L, M) = \begin{cases} \mathsf{value}_{\mathrm{B}}(L, M) & \mathsf{value}_{\mathrm{B}}(L, M) \neq \mathsf{undef}, \\ \mathsf{value}_{\mathrm{T}}(L, M) & \text{otherwise.} \end{cases}$$

Example 1. Consider the trail $M = [\![x > 0,\ x \mapsto 1,\ y \mapsto 0,\ z > 0]\!]$. The model interpretation according to M is $\upsilon[M] = [x \mapsto 1,\ y \mapsto 0]$. Therefore we have that $\mathsf{value}_{\mathrm{T}}(x > 0, M) = \mathsf{value}_{\mathrm{B}}(x > 0, M) = \mathsf{true}$, $\mathsf{value}_{\mathrm{T}}(x > 1, M) = \mathsf{false}$, $\mathsf{value}_{\mathrm{T}}(z > 0, M) = \mathsf{undef}$, $\mathsf{value}_{\mathrm{B}}(z > 0, M) = \mathsf{true}$, $\upsilon[M](x + y + 1) = 2$, and $\upsilon[M](x + z) = \mathsf{undef}$. The trail M is consistent, but $M' = [\![M, y < 0]\!]$ is not because $\mathsf{value}_{\mathrm{T}}(y < 0, M') = \mathsf{false}$ and $\mathsf{value}_{\mathrm{B}}(y < 0, M') = \mathsf{true}$. The trail M is not complete as it does not interpret z and therefore $\mathsf{value}_{\mathrm{T}}(z > 0, M) = \mathsf{undef}$. Finally, $M'' = [\![M, z < x]\!]$ is infeasible because $\{x \mapsto 1,\ z > 0,\ z < x\}$ is unsatisfiable, but M'' is consistent.

We extend the definition of value to clauses so that $\mathsf{value}(C, M) = \mathsf{true}$ if at least one literal of C evaluates to true, $\mathsf{value}(C, M) = \mathsf{false}$ if all literals evaluate to false, and $\mathsf{value}(C, M) = \mathsf{undef}$ otherwise. We say a clause C is satisfied by trail M if $\mathsf{value}(C, M) = \mathsf{true}$. A set of clauses $\mathcal{C}$ is satisfied by M if M is complete (and therefore consistent), and all clauses $C \in \mathcal{C}$ are satisfied by M. We use the predicate $\mathsf{satisfied}(\mathcal{C}, M)$ to denote that $\mathcal{C}$ is satisfied by M,

Given a set of clauses $\mathcal{C}_0$, our procedure starts with the state $\langle [\![]\!], \mathcal{C}_0 \rangle$ and performs transitions according to the rules we explain below. The goal is to either enter into a state sat denoting that the problem is satisfiable, or into a state unsat denoting that the problem is unsatisfiable. The states we traverse are either *search states* of the form $\langle M, \mathcal{C} \rangle$ or *conflict resolution states* of the form $\langle M, \mathcal{C} \rangle \vdash C$. In both types of states we keep the invariant that M is a consistent trail and $\mathcal{C}_0 \subseteq \mathcal{C}$. Additionally, in conflict resolution states the clause C is always a clause implied by $\mathcal{C}_0$ and refuted by the trail, i.e. $\mathcal{C}_0 \models C$ and $\mathsf{value}(C) = \mathsf{false}$. We call the clause C the *conflicting clause*.

To ensure termination, the transition system assumes existence of a finite set of literals $\mathbb{B}$ that we call the *finite basis*. During a derivation of the system, any literal added to the trail will be from $\mathbb{B}$, and the clauses that the system uses will only contain literals from $\mathbb{B}$.[2] The minimal assumption is that $\mathbb{B}$ must include all literals (and their negations) from the initial problem $\mathcal{C}_0$, and the theory-specific decision procedure must ensure that for any $\mathcal{C}_0$ such a finite basis exists.

2.1 Clausal Rules

We start by presenting the set of search rules and conflict analysis rules that resemble those of abstract DPLL and are the backbone of CDCL-style SAT

[2] Our finite basis corresponds to the closure of the literal-generating function used in splitting-on-demand [1].

DECIDE		
$\langle M, \mathcal{C}\rangle \longrightarrow \langle [\![M, L]\!], \mathcal{C}\rangle$	**if**	$L \in \mathbb{B}$, $\mathsf{value}(L, M) = \mathsf{undef}$
PROPAGATE		
$\langle M, \mathcal{C}\rangle \longrightarrow \langle [\![M, C{\rightarrow}L]\!], \mathcal{C}\rangle$	**if**	$C = (L_1 \vee \ldots \vee L_n \vee L) \in \mathcal{C}$ $\forall i : \mathsf{value}(L_i, M) = \mathsf{false}$ $\mathsf{value}(L, M) = \mathsf{undef}$
CONFLICT		
$\langle M, \mathcal{C}\rangle \longrightarrow \langle M, \mathcal{C}\rangle \vdash C$	**if**	$C \in \mathcal{C}$, $\mathsf{value}(C) = \mathsf{false}$
SAT		
$\langle M, \mathcal{C}\rangle \longrightarrow \mathsf{sat}$	**if**	$\mathsf{satisfied}(\mathcal{C}, M)$
FORGET		
$\langle M, \mathcal{C}\rangle \longrightarrow \langle M, \mathcal{C} \setminus \{C\}\rangle$	**if**	$C \in \mathcal{C}$ is a learned clause.

Fig. 1. Clausal search rules

solvers. The clausal search rules are presented in Fig. 1 and the clausal conflict analysis rules are presented in Fig. 2.

Search Rules. The DECIDE rule can take any literal from the basis $\mathbb{B}$ that does not yet have a value added to the trail. The PROPAGATE performs Boolean clause propagation by assigning to true the only unassigned literal from a clause where all other literals already evaluate to false.[3] If we encounter a clause C such all literals in C evaluate to false, we use the CONFLICT to enter conflict resolution state. During conflict analysis we can learn new clauses, and these can be removed using the FORGET rule. If our trail is complete and satisfies all the clauses, we can finish the search using the SAT rule.

Conflict Anaysis Rules. As in CDCL, the conflict analysis rules recover from a conflict encountered during the search, learn the reason of the conflict, and backtrack to an appropriate state to continue the search. The main analysis rule is the RESOLVE rule. This rule performs Boolean resolution of clauses C and D over the literal L, obtaining the clause $R = \mathsf{resolve}(C, D, L)$. The clause R is a valid deduction and moreover evaluates to false in M. If the result of the resolution is an empty clause (denoted with false), we can deduce that the problem is unsatisfiable using the UNSAT rule. Since, in a conflict analysis state $\langle M, \mathcal{C}\rangle \vdash C$, the clause C is a always a valid deduction, we can use the LEARN to add the clause C to set of clauses in order to aid in reducing the search space in the future. If this learned clause is at a later point not deemed useful, we can remove it using the FORGET rule. The CONSUME rules skips over those decided and propagated literals in the trail that are not relevant for the inconsistency of

[3] Note that in both DECIDE and PROPAGATE we do not need to ensure that the new literal does not cause an infeasibility. Additionally, we can decide literals that do not yet exist in $\mathcal{C}$, enabling refinement decisions as found in [6].

RESOLVE				
$\langle [\![M, D{\rightarrow}L]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow$	$\langle M, \mathcal{C}\rangle \vdash R$	**if**	$\neg L \in C$, $R = \mathsf{resolve}(C, D, L)$
CONSUME				
$\langle [\![M, D{\rightarrow}L]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow$	$\langle M, \mathcal{C}\rangle \vdash C$	**if**	$\neg L \notin C$
$\langle [\![M, L]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow$	$\langle M, \mathcal{C}\rangle \vdash C$	**if**	$\neg L \notin C$
BACKJUMP				
$\langle [\![M, N]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow$	$\langle [\![M, C{\rightarrow}L]\!], \mathcal{C}\rangle$	**if**	$C = L_1 \vee \ldots \vee L_m \vee L$, $\forall i : \mathsf{value}(L_i, M) = \mathsf{false}$, $\mathsf{value}(L, M) = \mathsf{undef}$, N starts with a decision
UNSAT				
$\langle M, \mathcal{C}\rangle \vdash \mathsf{false}$	$\longrightarrow$	unsat		
LEARN				
$\langle M, \mathcal{C}\rangle \vdash C$	$\longrightarrow$	$\langle M, \mathcal{C} \cup \{C\}\rangle \vdash C$	**if**	$C \notin \mathcal{C}$

Fig. 2. Clausal conflict analysis rules

the conflicting clause. Finally, we exit conflict resolution using the BACKJUMP rule, if we have deduced a clause C such that implies a literal earlier in the trail, skipping over at least one decision.

2.2 Theory-Specific Rules

We now extend the rules to enable theory-specific reasoning, allowing deductions in the style of DPLL(T), but more flexible, and allowing for assignments of variables to particular concrete values (Figure 3). As in DPLL(T), the basic requirement for a theory decision procedure is to provide an explain function that can explain theory-specific propagations and infeasible states. In DPLL(T), the theory explanations are theory lemmas in form of clauses that only contain negations of literals asserted so far. The explanation function explain here has more flexibility. Given a literal L and a consistent trail M that implies L to be true, i.e. such that $[\![M, \neg L]\!]$ is infeasible, $\mathsf{explain}(L, M)$ must return a valid theory lemma $E = L_1 \vee \ldots L_k \vee L$. The literals of the clause E must be from the finite basis $\mathbb{B}$, and all literals L_i must evaluate to false in M. Allowing explanations containing more than just the literals off the trail allows for more expressive lemmas and is crucial for model-based decision procedures. Limiting the literals to a finite basis, on the other hand, is important for ensuring the termination of the procedure.

Search Rules. We can propagate a literal L using the T-PROPAGATE rule, if it is implied in the current state, i.e. if adding the literal $\neg L$ to the trail M makes it infeasible. The side condition $\mathsf{value}(L, M) = \mathsf{undef}$ ensures that we cannot

T-Propagate

$$\langle M, \mathcal{C}\rangle \longrightarrow \langle [\![M, E{\rightarrow}L]\!], \mathcal{C}\rangle \quad \textbf{if} \quad \begin{array}{l} L \in \mathbb{B},\ \ \mathsf{value}(L, M) = \mathsf{undef} \\ \mathsf{infeasible}([\![M, \neg L]\!]) \\ E = \mathsf{explain}([\![M, \neg L]\!]) \end{array}$$

T-Decide

$$\langle M, \mathcal{C}\rangle \longrightarrow \langle [\![M, x{\mapsto}\alpha]\!], \mathcal{C}\rangle \quad \textbf{if} \quad \begin{array}{l} x \in \mathsf{vars}_T(\mathcal{C}) \\ \upsilon[M](x) = \mathsf{undef} \\ \mathsf{consistent}([\![M, x{\mapsto}\alpha]\!]) \end{array}$$

T-Conflict

$$\langle M, \mathcal{C}\rangle \longrightarrow \langle M, \mathcal{C}\rangle \vdash E \quad \textbf{if} \quad \begin{array}{l} \mathsf{infeasible}(M) \\ E = \mathsf{explain}(\mathsf{false}, M) \end{array}$$

T-Consume

$$\langle [\![M, x{\mapsto}\alpha]\!], \mathcal{C}\rangle \vdash C \longrightarrow \langle M, \mathcal{C}\rangle \vdash C \quad \textbf{if} \quad \mathsf{value}(C, M) = \mathsf{false}$$

T-Backjump-Decide

$$\langle [\![M, x{\mapsto}\alpha, N]\!], \mathcal{C}\rangle \vdash C \longrightarrow \langle [\![M, L]\!], \mathcal{C}\rangle \quad \textbf{if} \quad \begin{array}{l} C = L_1 \vee \ldots \vee L_m \vee L \\ \exists i : \mathsf{value}(L_i, M) = \mathsf{undef} \\ \mathsf{value}(L, M) = \mathsf{undef} \end{array}$$

Fig. 3. Theory search and conflict rules

produce an inconsistent trail ($\mathsf{value}(L, M) = \mathsf{false}$), nor include redundant information ($\mathsf{value}(L, M) = \mathsf{true}$). We use the T-Decide rule to assign an interpretation/value (α) to a variable x that occurs in our set of clauses ($x \in \mathsf{vars}_T(\mathcal{C})$). The side condition $\upsilon[M](x) = \mathsf{undef}$ ensures that we cannot assign a variable x that is already assigned in M, and $\mathsf{consistent}([\![M, x{\mapsto}\alpha]\!])$ ensures the new trail is consistent. We require a consistent trail because the function value is not well defined for inconsistent trails. We use the T-Conflict rule to enter conflict resolution state, whenever we detect that the trail is infeasible.

Conflict Analysis Rules. The T-Consume rule is similar to the consume rules in Figure 2, as it skips over a decided model assignment $x{\mapsto}\alpha$ that is not relevant for the inconsistency of the conflicting clause. The assignment is not relevant because the conflicting clause C still evaluates to false after the assignment $x \mapsto \alpha$ is removed from the trail. We use the T-Backjump-Decide rule when we reach an assignment $x \mapsto \alpha$ that is relevant for the inconsistency of the conflicting clause C, but the Backjump rule is not applicable because C contains more than one literal that evaluates to undef after we remove the assignment $x{\mapsto}\alpha$ from the trail. The T-Backjump-Decide rule may generates doubts about the termination argument for our abstract procedure, since it is just replacing a decision $x \mapsto \alpha$ with another decision L. In our termination proof, we justify that by assuming that Boolean decisions (L) have "bigger" weight than model assignment decisions ($x{\mapsto}\alpha$).

2.3 Producing Explanations

A crucial component of our framework is the explanation function explain. Given an infeasible trail M, it must be able to produce a valid theory lemma that is inconsistent with M using only literals from the finite basis.

If the infeasibility of M only depends on literals that already occur M, then explain can simply return the inconsistent clause $\neg L_1 \vee \ldots \vee \neg L_k$ where all literal L_i occur in M. For example, the trail $M = [\![x < 0,\ \ y > 1,\ \ x > 0]\!]$ is infeasible, and $\mathsf{explain}(\mathsf{false}, M) = \neg(x < 0) \vee \neg(x > 0)$ is a valid theory lemma that is inconsistent with M. In such cases, the T-Conflict and T-Propagate rules correspond to the theory propagation and conflict rules from the DPLL(T) framework.

The more interesting case occurs when the infeasibility on a trail M also depends on decided model assignments $x \mapsto \alpha$ in M. Consider, for example, the infeasible (but consistent) trail

$$M = [\![y > 0,\ \ z > 0,\ \ x + y + z < 0,\ \ x \mapsto 0]\!] \ .$$

It might be tempting to define $\mathsf{explain}(\mathsf{false}, M)$ to produce the valid theory lemma

$$\neg(y > 0) \vee \neg(z > 0) \vee \neg(x + y + z < 0) \vee \neg(x = 0)$$

This naïve explain function that just replaces the assignments $x \mapsto \alpha$ with literals $x = \alpha$, is inadequate as it *does not* satisfy the finite basis requirement for theories that operate over infinite domains, such as the Integers or the Reals. Using such a function, in this example, we would be able to continue indefinitely by assigning x to 1, then 2, and so on.

In principle, for any theory that admits elimination of quantifiers, it is possible to construct an explanation function explain that satisfy the finite basis requirement. The basic idea is to eliminate all unassigned variables and produce an implied formula that is also inconsistent with the assigned variables in the infeasible trail. In the previous example, the variables y and z are unassigned, so we can separate the infeasible literals that the naïve explain function return into the literals from the trail and literals from assignments

$$A \equiv (y > 0) \wedge (z > 0) \wedge (x + y + z < 0) \qquad\qquad B \equiv (x = 0) \ .$$

Given A, we use a quantifier elimination procedure to generate a CNF formula F of the form $C_1 \wedge \ldots \wedge C_k$ that is equivalent to $(\exists y, z : A)$, and therefore also inconsistent with B while only using the assigned variables (in this case x). Note that B corresponds to the assignment $v[M]$, and each clause C_i evaluates to true or false under $v[M]$ because all variables occurring in C_i are assigned by $v[M]$. Moreover, at least one of these clauses must evaluate to false because F is inconsistent with B. Let I be the clause C_i that evaluates to false. We have that $A \implies I$ is a valid clause because

$$A \implies (\exists y, z : A) \iff F \implies I$$

Since I is inconsistent with B, the clause $A \implies I$ will also be inconsistent with the trail and can be used as an explanation. Moreover, I is an *interpolant*

for A and B. In this example, we obtain $0 < x$ by eliminating the variables y and z from A using Fourier-Motzkin elimination, resulting in the explanation $A \implies (0 < x)$. When solving a set of linear arithmetic $\mathcal{C}$, Fourier-Motzkin elimination is sufficient to define the explain function, as shown in [12,9]. Fourier-Motzkin elimination gives a finite-basis $\mathbb{B}$ with respect to $\mathcal{C}$, and the basis can be obtained by closing $\mathcal{C}$ under the application of Fourier-Motzkin elimination step. It is fairly easy to show that the closure is a finite set, since we always produce constraints with one variable less.

In the last example, $0 < x$ is an *interpolant* for A and B, but so is $x \neq 0$. The key point is that an arbitrary interpolation procedure does not guarantee a finite basis. Nonetheless, this observation is useful when designing explanation functions for more complex theories. For nonlinear arithmetic constraints, we describe how to produce an explain procedure that produces an interpolant based on cylindrical algebraic decomposition (CAD) in [8]. The theory of uninterpreted functions (UF) does not admit quantifier elimination, but *dynamic-ackermannization* [5] can be used to create an interpolant I between A and B without compromising the finite basis requirement. For example, the trail $M = [\![x \mapsto 0,\ \ y \mapsto 0,\ \ f(x) \neq f(y)]\!]$ is infeasible with respect to UF. Then, we have $A \equiv f(x) \neq f(y)$ and $B \equiv (x = 0) \wedge (y = 0)$. Using dynamic-ackermannization we have that $I \equiv x \neq y$ is an interpolant for A and B, and $f(x) \neq f(y) \implies x \neq y$ is a valid theory lemma that is also inconsistent with M. We satisfy the finite basis requirement because dynamic-ackermannization only "introduces" equalities between terms that already exist in the trail. Finally, an explain function for the theory of arrays can be built on top of the approach described in [3].

Example 2. For the sake of simplicity, we restrict ourselves to the case of linear arithmetic. We illustrate the search rules by applying them to the following clauses over Boolean and linear arithmetic atoms $\mathcal{C} = \{x < 1 \vee p,\ \ \neg p \vee x = 2\}$. We start the deduction from the initial state $\langle [\![]\!], \mathcal{C}\rangle$ and apply the rules of the mcSAT system.

$\langle [\![]\!], \mathcal{C}\rangle$
$\downarrow$ T-Decide $(x \mapsto 1)$
$\langle [\![x \mapsto 1]\!], \mathcal{C}\rangle$
$\downarrow$ Propagate (p must be true, since $x < 1$ evaluates to false in the current trail)
$\langle [\![x \mapsto 1,\ \ (x < 1 \vee p) \to p]\!], \mathcal{C}\rangle$
$\downarrow$ Conflict ($\neg p$ and $x = 2$ evaluate to false in the current trail)
$\langle [\![x \mapsto 1,\ \ (x < 1 \vee p) \to p]\!], \mathcal{C}\rangle \vdash \neg p \vee x = 2$
$\downarrow$ Resolve (resolving $x < 1 \vee p$ and $\neg p \vee x = 2$)
$\langle [\![x \mapsto 1]\!], \mathcal{C}\rangle \vdash x < 1 \vee x = 2$

In this state, the Backjump rule is not applicable because in the conflicting clause, both $x < 1$ and $x = 2$ evaluate to undef after the model assignment $x \mapsto 1$ is removed from the trail. The intuition is that the model assignment was "premature", and the clause $x < 1 \vee x = 2$ is indicating that we should decide $x < 1$ or $x = 2$ before we assign x.

$\langle [\![x \mapsto 1]\!], \mathcal{C} \rangle \vdash x < 1 \vee x = 2$
$\downarrow$ T-Backjump-Decide
$\langle [\![x < 1]\!], \mathcal{C} \rangle$
$\downarrow$ T-Decide (the model assignment must satisfy $x < 1$)
$\langle [\![x < 1,\ \ x \mapsto 0]\!], \mathcal{C} \rangle$
$\downarrow$ Propagate ($\neg p$ must be true, since $x = 2$ evaluates to false in the current trail)
$\langle [\![x < 1,\ \ x \mapsto 0,\ \ (\neg p \vee x = 2) \rightarrow \neg p]\!], \mathcal{C} \rangle$
$\downarrow$ Sat ($x \mapsto 0$ and $\neg p$ satisfy all clauses)
sat

Example 3. Now, we consider a set of linear arithmetic (unit) clauses

$$\mathcal{C} = \{ x < 1,\ \ x < y,\ \ 1 < z,\ \ z < x \} \ .$$

To simplify the presentation of this example, with a small abuse of notation, we use $\hookrightarrow L$ instead of $L \rightarrow L$, whenever the literal L is implied by the unit clause L.

$\langle [\![ ]\!], \mathcal{C} \rangle$
$\downarrow$ Propagate $\times$ 4 (propagate all unit clauses)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x]\!], \mathcal{C} \rangle$
$\downarrow$ T-Decide (the current trail is consistent with the model assignment $x \mapsto 0$)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ x \mapsto 0]\!], \mathcal{C} \rangle$
$\downarrow$ T-Decide (peek a value for y, keeping consistency, y s.t. $x < y$)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ x \mapsto 0,\ y \mapsto 1]\!], \mathcal{C} \rangle$
$\downarrow$ T-Conflict ($1 < z$ and $z < x$ implies that $1 < x$)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ x \mapsto 0,\ y \mapsto 1]\!], \mathcal{C} \rangle \vdash C$

In the state above, the conflict was detected by noticing that we can not pick a value for z, because the trail contains $1 < z$, $z < x$ and $x \mapsto 0$. The explain procedure "generates" the explanation clause $C \equiv \neg(1 < z) \vee \neg(z < x) \vee 1 < x$ by eliminating z using Fourier-Motzkin elimination, and this clause evaluates to false in the current trail. Note that, as in DPLL(T), we could have also explained the infeasibility trail by producing the clause $\neg(1 < z) \vee \neg(z < x) \vee \neg(x < 1)$ that uses only literals that already exist in the trail. However, this is clause is weaker since $\neg(x < 1) \equiv (1 \leq x)$ is weaker than $1 < x$. We continue the deduction by analyzing the conflict.

$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ x \mapsto 0,\ y \mapsto 1]\!], \mathcal{C} \rangle \vdash C$
$\downarrow$ T-Consume (the conflict does not depend on y)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ x \mapsto 0]\!], \mathcal{C} \rangle \vdash C$
$\downarrow$ Backjump (after backtracking $x \mapsto 0$, the clause C implies $1 < x$)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ C \rightarrow 1 < x]\!], \mathcal{C} \rangle$

After the application of the backjump rule, the newly asserted literal $1 < x$ is immediately in conflict with the literal $x < 1$ and we enter conflict resolution again, with the explanation obtained by Fourier-Motzkin elimination.

$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ C \hookrightarrow 1 < x]\!], \mathcal{C} \rangle$
$\downarrow$ T-Conflict $(1 < x$ and $x < 1$ implies $\mathsf{false})$
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x,\ C \hookrightarrow 1 < x]\!], \mathcal{C} \rangle \vdash \neg(1 < x) \vee \neg(x < 1)$
$\downarrow$ Resolve $\times 3$, Consume, Resolve, Unsat
unsat

Theorem 1. *Given a set of clauses $\mathcal{C}$, and assuming a finite basis explanation function* explain*, any derivation starting from the initial state $\langle [\![]\!], \mathcal{C} \rangle$ will terminate either in a state* sat*, when $\mathcal{C}$ is satisfiable, or in the* unsat *state. In the later case, the set of clauses $\mathcal{C}$ is unsatisfiable.*

Proof. Assume we have a set of clauses $\mathcal{C}$, over the variables $x_1, \ldots, x_n$, and a finite-basis explanation function explain. Starting from the initial state $\langle [\![]\!], \mathcal{C} \rangle$, we claim that any derivation of the transition system (finite or infinite), satisfies the following properties

1. the only possible "sink states" are the sat and the unsat states;
2. all $\vdash C$ clauses are implied by the initial set of clauses $\mathcal{C}$;
3. during conflict analysis $\mathsf{value}(C, M) = \mathsf{false}$ for the $\vdash C$ clauses;

Assuming termination, and the above properties, the statement can be proven easily. Since sat and unsat are the only sink states, the derivation will terminate in one of these states. Since the Sat rule is only applicable if the set of clauses $\mathcal{C}$ is satisfied, we have that the original problem is indeed satisfiable. On the other hand, if we terminate in the unsat state, by the above properties, the conflicting clause false is implied by the inital $\mathcal{C}$. Given that false is implied by the original set of clauses, the initial clauses themselves must truly be unsatisfiable.

The first property in the list above is a fairly easy exercise in case analysis and induction, so we skip those and concentrate on the more interesting properties. Proving the properties of conflict analysis is also quite straightforward, via induction on the number of conflicts, and conflict analysis steps. Clearly, initially, we have that C evaluates to false (the precondition of the Conflict and T-Conflict rules), and is implied by $\mathcal{C}$ by induction. Then, every new clause that we produce during conflict resolution is obtained by the Boolean resolve rule, which will produce a valid deduction. Additionally, since the clause we are resolving with is a proper explanation, it will have all literals except the one we are resolving evaluate to false. Therefore, the resolvent also evaluates to false. As we backtrack down the trail with the conflicting clause, by definition of value and the preconditions of the rules, the clause still remains false.

Now, let us prove that the system terminates. It is clear that the conflict analysis rules (always removing elements from the trail) always terminate in a finite number of steps, and return to the search rules (or the unsat state). For the sake of the argument, let us assume that there is a derivation that does not terminate, and therefore does not enter the unsat state. We can define a big-step transition relation $\longrightarrow_{\text{bs}}$ that covers a transition from a search state, applying one or more transitions in the conflict analysis rules, and returns to a search state.

By assumption, we have a finite-basis explanation function explain, so we can assume a set of literals $\mathbb{B}$ from which all the clauses that we can see during the search are constructed. In order to keep progress of the search, we define a partial order $M_1 \prec M_2$ on trails. The trail contains three different kinds of element: model assignments decisions ($x \mapsto \alpha$), Boolean decisions (L) and propagations ($C \to L$). The basic idea is to consider that propagations are heavier than Boolean decisions that are heavier than model assignments. We capture that by defining a (weight) function w from trail elements into the set $\{1, 2, 3\}$.

$$w(C \to L) = 3, \quad w(L) = 2, \quad w(x \mapsto \alpha) = 1$$

We define the $M_1 \prec M_2$ using a lexicographical order based on the weights of the trail elements, i.e.

$$\begin{aligned} [\![]\!] \prec M &= \mathsf{true} \ \textbf{if} \ M \neq [\![]\!] \\ M \prec [\![]\!] &= \mathsf{false} \\ [\![a, M_1]\!] \prec [\![b, M_2]\!] &= w(a) < w(b) \vee (w(a) = w(b) \wedge M_1 \prec M_2) \end{aligned}$$

It is clear that $[\![]\!]$ is the minimal element, and any trail containing $|\mathbb{B}|$ propagations followed by n model assignments is maximal. It is easy to see that for all trails M and trail elements a we have that $M \prec [\![M, a]\!]$. Thus, any rule that adds a new element to the trail is essentially creating a "bigger" trail with respect to the partial order $\prec$. This simple observation covers most of our rules.

Now, we consider the big-step $\longrightarrow_{\text{bs}}$ transition from a state $\langle M_1, \mathcal{C}_1 \rangle$ into a state $\langle M_2, \mathcal{C}_2 \rangle$. If we return using Backjump, then the trail M_1 is of the form $[\![M, L, N]\!]$ or $[\![M, x \mapsto \alpha, N]\!]$, and the trail M_2 is of the form $[\![M, C \to L']\!]$. In both cases $M_1 \prec M_2$ because a $w(C \to L')$ is greater than $w(L)$ and $w(x \mapsto \alpha)$. Similarly, if we return using T-Backjump-Decide, the trail M_1 is of the form $[\![M, x \mapsto \alpha, N]\!]$, and M_2 is of the form $[\![M, L]\!]$, once again $M_1 \prec M_2$ since $w(L) > w(x \mapsto \alpha)$.

Now, to justify the Forget rule, we define a partial order $\langle M_1, \mathcal{C}_1 \rangle \lessdot \langle M_2, \mathcal{C}_2 \rangle$ as $M_1 \prec M_2 \vee (M_1 = M_2 \wedge |\mathcal{C}_1| > |\mathcal{C}_2|)$. Using this definition, we have that $\langle M, \mathcal{C} \rangle \lessdot \langle M, \mathcal{C} \setminus \{C\} \rangle$, and consequently the Forget rule also produces a "bigger" state. The partial order $\lessdot$ also have maximal elements $\langle M_{\mathsf{max}}, \mathcal{C}_0 \rangle$, where M_{max} is a maximal element for $\prec$ and $\mathcal{C}_0$ is the initial set of clauses. Since all rules are producing bigger states and we can not increase forever, the termination of the system follows. □

3 Conclusion

We proposed a model-constructing satisfiability calculus (mcSAT) that encompasses all model-based decision procedures found in the SMT literature. The mcSAT framework extends DPLL(T) by allowing assignments of variables to concrete values and relaxing the restriction on creation of new literals. The model created during the search also takes part in explaining the conflicts, and the full model is readily available as a witness if the procedure reports a satisfiable answer. The new calculus also extends nlsat, proposed at [8], by removing unnecessary restrictions on the Boolean-level search rules – it allows efficient Boolean

constraint propagation found in state-of-the-art SAT solvers and incorporates theory propagation and conflict rules from the DPLL(T) framework. The mcSAT calculus allows SMT developers to combine existing and successful techniques from the DPLL(T) framework with the flexibility provided by the nlsat calculus. We also presented a correctness proof for our procedure that is much simpler than the one provided for nlsat.

In this article, we did not explore the theory combination problem. However, we believe mcSAT is the ideal framework for combining implementations for theories such as: arithmetic, bit-vectors, floating-point, uninterpreted functions, arrays and datatypes.

References

1. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on Demand in SAT Modulo Theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability. IOS Press (2009)
3. Brummayer, R., Biere, A.: Lemmas on Demand for the Extensional Theory of Arrays. Journal on Satisfiability, Boolean Modeling and Computation 6 (2009)
4. Cotton, S.: Natural Domain SMT: A Preliminary Assessment. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 77–91. Springer, Heidelberg (2010)
5. de Moura, L., Bjørner, N.: Model-based Theory Combination. In: Satisfiability Modulo Theories. ENTCS, vol. 198, pp. 37–49 (2008)
6. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding floating-point logic with systematic abstraction. In: FMCAD (2012)
7. Jovanović, D., de Moura, L.: Cutting to the Chase Solving Linear Integer Arithmetic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 338–353. Springer, Heidelberg (2011)
8. Jovanović, D., de Moura, L.: Solving Non-linear Arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
9. Korovin, K., Tsiskaridze, N., Voronkov, A.: Conflict Resolution. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 509–523. Springer, Heidelberg (2009)
10. Krstić, S., Goel, A.: Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
11. Malik, S., Zhang, L.: Boolean satisfiability from theoretical hardness to practical success. Communications of the ACM 52(8), 76–82 (2009)
12. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to Richer Logics. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC (2001)
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
15. Silva, J.P.M., Sakallah, K.A.: GRASP – a new search algorithm for satisfiability. In: ICCAD (1997)

Automata as Proofs

Andreas Podelski

University of Freiburg, Germany

Abstract. A recent approach to the verification of programs constructs a correctness proof in the form of a finite automaton. The automaton recognizes a set of traces. Here, a trace is any sequence of statements (not necessarily feasible and not necessarily on a path in the control flow graph of the program). A trace can be formalized as a word over the alphabet of statements. A trace can also be viewed as as special case of a program. Applying static analysis or a symbolic method (e.g., SMT solving with interpolant generation) to a single trace τ, a correctness proof for the trace τ can be obtained in the form of a sequence of consecutive Hoare triples (or, phrased differently, an inductive sequence of assertions). We can construct an automaton that contains a transition $q_\varphi \xrightarrow{a} q_{\varphi'}$ for each Hoare triple $\{\varphi\}a\{\varphi'\}$ in the correctness proof for the trace τ. The automaton accepts the trace τ. In fact, the automaton accepts all traces whose correctness proof uses the same set of Hoare triples as the trace τ. Given a program and a number of traces $\tau_1, \ldots, \tau_n$ of the program, we can construct an automaton from the n different correctness proofs for the traces $\tau_1, \ldots, \tau_n$. The automaton recognizes a set of correct traces. We still need to check whether this set includes all the traces on a path in the control flow graph of the program. The check is an automata-theoretic operation (which is reducible to non-reachability in a finite graph). That is, the two steps of constructing and checking a proof neatly separate the two concerns of data and control in program verification. The construction of a proof in the form of an automaton accounts for the interpretation of statements in data domains. The automaton, however, has a meaning that is oblivious to the interpretation of statements: a set of words over a particular alphabet. The check of the proof uses this meaning of the automaton and accounts for the control flow of the program. The implementation of the check of the proof as an automata-theoretic inclusion check is reminiscent of model checking (the finite control flow graph defines the model, the automaton defines the property). The resulting verification method is not compositional in the syntax of the program; it is compositional in a new, semantics-directed sense where modules are sets of traces; the sets of traces are constructed from mutually independent correctness proofs and intuitively correspond to different *cases* of program executions. Depending on the verification problem (the correctness property being safety or termination for sequential, recursive, or concurrent programs), the approach uses non-deterministic automata, nested-word automata, Büchi automata, or alternating automata as proofs; see, e.g., [2,3,1].

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 13–14, 2013.

References

1. Farzan, A., Podelski, A., Kincaid, Z.: Inductive Data Flow Graphs. In: Cousot, R., Giacobazzi, R. (eds.) POPL. ACM (to appear, 2013)
2. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of Trace Abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
3. Heizmann, M., Hoenicke, J., Podelski, A.: Nested Interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 471–482. ACM (2010)

Complete Abstractions Everywhere

Francesco Ranzato

Dipartimento di Matematica, University of Padova, Italy

Abstract. While soundness captures an essential requirement of the intrinsic approximation of any static analysis, completeness encodes approximations that are as precise as possible. Although a static analysis of some undecidable program property cannot be complete relatively to its reference semantics, it may well happen that it is complete relatively to an approximated and decidable reference semantics. In this paper, we will argue on the ubiquity of completeness properties in static analysis and we will discuss the beneficial role that completeness can play as a tool for designing and fine-tuning static analyses by reasoning on the completeness properties of their underlying abstract domains.

1 Introduction

Static analysis relies on sound (a.k.a. correct) and computable semantic approximations. A system $\mathcal{S}$ (or some observable behavior of it) is modeled by some semantics $\mathrm{Sem}[\![\mathcal{S}]\!]$ and a static analysis of $\mathcal{S}$ is designed as an approximate semantics $\mathrm{Sem}^\sharp[\![\mathcal{S}]\!]$ which must be sound w.r.t. $\mathrm{Sem}[\![\mathcal{S}]\!]$. This may be called global soundness of static analysis. A system $\mathcal{S}$ is typically designed by some form of composition of a number of constituents c_i and this is reflected on its global semantics $\mathrm{Sem}[\![\mathcal{S}]\!]$ which is commonly defined by some combinations of the semantics $\mathrm{Sem}[\![c_i]\!]$ of its components. Thus, global soundness of a static analysis of $\mathcal{S}$ is ordinarily derived from local soundness of static analyses for its components c_i. This global vs. local picture of the soundness of static analysis is very common, independently of the kind of systems (e.g., programs, reactive systems, hardware systems), of static analysis techniques (dataflow analysis, model checking, abstract interpretation, logical deductive systems, type systems, etc.), of system properties under analysis (safety, liveness, numerical properties, pointer aliasing, type safety, etc.). We might single out a basic and rough proof principle in static analysis: global soundness is derived from local soundness. In particular this applies to static analyses that are designed using some form of abstract interpretation. Let us consider a simplified and recurrent scenario. A constituent of the global semantics is defined as least (or greatest) fixpoint (denoted by lfp) of a monotone function f on some domain of properties D which is endowed with a partial order that encodes relative precision of properties. Here, $\mathrm{lfp}(f)$ play the role of global semantics which is therefore defined through a component $f : D \to D$ that plays the role of local semantics (e.g., f is a transfer function in program analysis). In abstract interpretation, a static analysis is then specified as an abstract fixpoint computation which must be sound for $\mathrm{lfp}(f)$. This is routinely defined through an ordered abstract domain A of properties and an abstract semantic function $f^\sharp : A \to A$ that give rise to a fixpoint-based static analysis $\mathrm{lfp}(f^\sharp)$ (whose decidability and/or practical scalability is usually ensured by chain conditions on A,

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 15–26, 2013.

widenings/narrowings, interpolations, etc.). Soundness relies on encoding approximation through a concretization map $\gamma : A \to D$ and/or an abstraction map $\alpha : D \to A$: the approximation of some value d through an abstract property a is encoded as $\alpha(d) \leq_A a$ or — equivalently when α/γ form a Galois connection — $d \leq_D \gamma(a)$. Hence, global soundness translates to $\alpha(\mathrm{lfp}(f)) \leq \mathrm{lfp}(f^\sharp)$, local soundness means $\alpha \circ f \leq f^\sharp \circ \alpha$, and the well-known "fixpoint approximation lemma" tells us that local implies global soundness. This fixpoint approximation lemma has been first used in the early abstract interpretation papers [8, Section 9], [6, Chapter 5], [9, Theorem 7.1.0.4], and has been later rediscovered a number of times (e.g., [1, Section 3]).

While (both global and local) soundness captures a basic requirement of any reasonable semantic approximation, completeness encodes approximations that are as precise as possible. Once an abstract domain of properties A has been chosen, A yields a strict upper bound to the precision of static analyses that approximate $\mathrm{Sem}[\![\mathcal{S}]\!]$ over A: this means that $\alpha(\mathrm{Sem}[\![\mathcal{S}]\!])$ represents the best approximation that one can achieve over A. We refer to as *completeness* when $\alpha(\mathrm{Sem}[\![\mathcal{S}]\!]) = \mathrm{Sem}^\sharp[\![\mathcal{S}]\!]$ happens. In fixpoint-based semantics, an abstract interpretation is globally complete when $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^\sharp)$. Locally for f, completeness means that $\alpha \circ f = f^\sharp \circ \alpha$, meaning that in each local step, $f^\sharp$ behaves in the best possible way as allowed by A. The approximation lemma for soundness becomes the "transfer lemma" for completeness stating the proof principle that local completeness implies global completeness. To the best of our knowledge, this transfer lemma first appears in the early abstract interpretation paper [9, Theorems 7.1.0.4], and has been later re-stated and re-proved multiple times, e.g., [3, Fact 2.3], [4, Lemma 4.3], [1, Section 3] (where it is called μ-fusion rule).

Of course, completeness cannot happen when D is able to represent undecidable properties of $\mathcal{S}$ and the static analysis $\mathrm{Sem}^\sharp[\![\mathcal{S}]\!]$ is computable. Completeness may instead occur and is actually quite common in comparative semantic modeling, i.e., in studying semantic models at different levels of abstraction. For instance, Cousot [7] defines a comprehensive abstract interpretation-based hierarchy of the most common program semantics and highlights where completeness holds. It is important to remark that although a static analysis cannot be complete relatively to its reference semantics, it may well happen that it is complete relatively to an approximated and decidable reference semantics, e.g., as a notable case, a different and more precise static analysis. In this paper, we will argue the important role that completeness can play as a tool for designing and fine-tuning static analyses by reasoning on the completeness properties of their underlying abstract domains. We will provide a number of examples ranging from static analysis to model checking where the view of an abstract domain A under a "completeness light" allows us to understand the deep reason of why and how A actually works.

2 Complete and Exact Abstractions

As mentioned above, static program analysis will be here formalized in a simplified abstract interpretation scenario.

Program Semantics. We consider simple while programs with integer variables in *Var*, integer arithmetic expressions in *Aexp* and boolean expressions in *Bexp*. Hence,

$\mathrm{Store} \triangleq \mathit{Var} \to \wp(\mathbb{Z})$ denotes the set of stores, $[\![e]\!] : \mathrm{Store} \to \mathbb{Z}$ the semantics of an arithmetic expression e and $[\![b]\!] : \mathrm{Store} \to \{\mathbf{f}, \mathbf{t}\}$ the semantics of a boolean expression b. Transfer functions $(\!|\cdot|\!) : \wp(\mathrm{Store}) \to \wp(\mathrm{Store})$ model how assignments $x := e$ and boolean tests b affect the store: $(\!|x := e|\!)S \triangleq \{\rho[x \mapsto [\![e]\!]\rho] \in \mathrm{Store} \mid \rho \in S\}$ and $(\!|b|\!)S \triangleq \{\rho \in S \mid [\![b]\!]\rho = \mathbf{t}\}$. The program semantics is usually defined as least fixpoint of a system of recursive equations indexed on program points. We consider here a denotational-style program semantics $[\![P]\!] : \wp(\mathrm{Store}) \to \wp(\mathrm{Store})$ whose definition is standard:

$$[\![stm_1;\ stm_2]\!]S \triangleq [\![stm_2]\!]([\![stm_1]\!]S), \quad [\![\textbf{if } b \textbf{ then } stm]\!]S \triangleq [\![stm]\!](\!|b|\!)S \cup (\!|\neg b|\!)S,$$
$$[\![\textbf{while } b \textbf{ do } stm]\!]S \triangleq (\!|\neg b|\!)\big(\mathrm{lfp}(\lambda T.S \cup [\![stm]\!](\!|b|\!)T)\big).$$

Soundness, Completeness and Exactness. A static program analysis is defined by an abstract semantics of while programs that relies on some store *abstraction*. Given a generic domain of properties D, we denote by $\mathrm{Abs}(D)$ the class of all possible abstractions of D. Thus, given an abstraction $A \in \mathrm{Abs}(\wp(\mathrm{Store}))$ of store properties, a static analysis needs to define *locally sound* abstract transfer functions on A and an abstract program semantics on A which relies on these abstract transfer functions and is required to be *globally sound*. Let us briefly recall how abstractions and local/global soundness are formalized in abstract interpretation.

Abstractions are formalized as domains in abstract interpretation. If $A \in \mathrm{Abs}(D)$ then both D and A are partially ordered by an approximation ordering where $x \leq y$ means that y approximates x (or x is more precise than y). D and A are required to be complete lattices in order to have arbitrary lub's that model concrete and abstract disjunctions. Finally, the abstraction A is related to D at least by a monotone concretization function $\gamma : A \to D$ which provides the concrete meaning of abstract values, so that: (1) $d \leq_D \gamma(a)$ means that a is an abstract approximation of d and (2) $f \circ \gamma \leq_D \gamma \circ f^\sharp$ means that the abstract function $f^\sharp : A \to A$ is a correct (or sound) approximation of the concrete function $f : D \to D$. A concretization function does not ensure the existence of best abstractions in A for concrete properties, i.e., it is not guaranteed that $\wedge_A\{a \in A \mid d \leq_D \gamma(a)\}$ is an abstract approximation of d. Consequently, a concretization function does not guarantee the existence of best correct approximations of concrete functions. Actually, most abstract domains A are related to D also by an abstraction function $\alpha : D \to A$ which provides the best approximation in A of any concrete property. Accordingly, $f^\sharp$ is a correct approximation of f when $\alpha \circ f \leq f^\sharp \circ \alpha$. Clearly, γ and α must agree on the way they encode approximation: this translates to $\alpha(d) \leq_A a \Leftrightarrow d \leq_D \gamma(a)$ and amounts to require that $\langle \alpha, \gamma \rangle$ forms a Galois connection. When f and $f^\sharp$ are monotonic, and therefore have least (and greatest) fixpoints, local soundness for $f^\sharp$ implies global fixpoint soundness $\alpha(\mathrm{lfp}(f)) \leq_A \mathrm{lfp}(f^\sharp)$.

If $f \circ \gamma = \gamma \circ f^\sharp$ then $f^\sharp$ is called an *exact approximation* of f, while $f^\sharp$ is called a *complete approximation* of f when $\alpha \circ f = f^\sharp \circ \alpha$ holds. Here, we have that local exactness/completeness implies global exactness/completeness: If $f^\sharp$ is exact then $\mathrm{gfp}(f) = \gamma(\mathrm{gfp}(f^\sharp))$ and if $f^\sharp$ is complete then $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^\sharp)$. These can be viewed as two facets of the principle "local implies global completeness" translated in abstract interpretation terms. Exactness and completeness are orthogonal notions. Let us point out that global exactness means that the abstract fixpoint computation $\mathrm{gfp}(f^\sharp)$

yields an exact representation in A of $\mathrm{gfp}(f)$ while global completeness guarantees that the abstract fixpoint computation $\mathrm{lfp}(f^\sharp)$ provides the best representation in A of $\mathrm{lfp}(f)$.

When a Galois connection is assumed, we have that $f^\sharp$ is sound iff $\alpha \circ f \circ \gamma \leq f^\sharp$. Thus, any function f has a best correct approximation in A which is, at least theoretically, defined as $f^{\mathrm{best}_A} \triangleq \alpha \circ f \circ \gamma$. Moreover, it turns out that the possibility of defining an exact or complete abstract function $f^\sharp$ on some abstraction A actually depends only on A itself. In fact, we have that $f^\sharp$ is complete iff $f^\sharp = f^{\mathrm{best}_A}$ and $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$, while $f^\sharp$ is exact iff $f^\sharp = f^{\mathrm{best}_A}$ and $f \circ \gamma = \gamma \circ \alpha \circ f \circ \gamma$. In the following, we thus make reference to a complete and exact abstraction A to mean completeness and exactness of f^{best_A}.

Abstract Program Semantics. Given a store abstraction $A \in \mathrm{Abs}(\mathrm{Store})$, abstract transfer functions $(\!|\, x := e \,|\!)^A : A \to A$ and $(\!|\, b \,|\!)^A : A \to A$ must be locally sound, i.e. correct: for any set S of stores, $\alpha((\!|\, x := e \,|\!)S) \leq_A (\!|\, x := e \,|\!)^A \alpha(S)$ and $\alpha((\!|\, b \,|\!)S) \leq_A (\!|\, b \,|\!)^A \alpha(S)$. Also observe that the best abstract transfer functions on A are as follows:

$$(\!|\, x := e \,|\!)^{\mathrm{best}_A} a = \alpha(\{\rho[x \mapsto [\![e]\!]\rho] \in \mathrm{Store} \mid \rho \in \gamma(a)\})$$
$$(\!|\, b \,|\!)^{\mathrm{best}_A} a = \alpha(\{\rho \in \gamma(a) \mid [\![b]\!]\rho = \mathbf{t}\})$$

The abstract denotational semantics $[\![P]\!]^A : A \to A$ is derived simply by composing abstract transfer functions and by approximating concrete disjunctions of stores through abstract lub's in A:

$$[\![stm_1;\ stm_2]\!]^A a \triangleq [\![stm_2]\!]^A([\![stm_1]\!]^A a)$$
$$[\![\mathbf{if}\ b\ \mathbf{then}\ stm]\!]^A a \triangleq [\![stm]\!]^A((\!|\, b \,|\!)^A a) \sqcup_A (\!|\, \neg b \,|\!)^A a$$
$$[\![\mathbf{while}\ b\ \mathbf{do}\ stm]\!]^A a \triangleq (\!|\, \neg b \,|\!)^A \big(\mathrm{lfp}(\lambda x.a \sqcup_A [\![stm]\!]^A (\!|\, b \,|\!)^A x)\big)$$

Of course, in the abstract fixpoint computation for a while-loop, a widening operator ∇_A may be used in place of the lub $\sqcup_A$ for abstractions A that do not satisfy the ascending chain condition or to accelerate convergence in A.

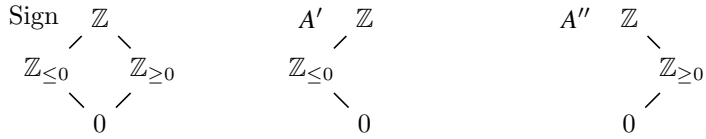
Soundness of the abstract semantics, that is $\alpha([\![P]\!]) \leq_A [\![P]\!]^A$, is a consequence of three main points:

(1) Local soundness of abstract functions is preserved by their composition;
(2) The abstract lub $\sqcup_A$ is a sound approximation of the union $\cup$;
(3) Local soundness implies global fixpoint soundness.

3 Completeness in Program Analysis

Given a program P with n integer variables, in order to simplify the notation, we view a n-tuple of integer values as a store for P so that $\langle \wp(\mathbb{Z}^n), \subseteq \rangle$ plays the role of concrete domain and store (either relational or non-relational) abstractions range in $\mathrm{Abs}(\wp(\mathbb{Z}^n))$. Let us consider the simplest case of one single integer variable. Observe that the trivial abstraction $A^\top = \{\mathbb{Z}\}$, which abstracts each set of integers to the top value $\mathbb{Z}$, is

always obviously complete, meaning that the mapping $\{\mathbb{Z} \mapsto \mathbb{Z}\}$ is both a complete abstract transfer function and a complete abstract program semantics. Let us therefore consider the simplest abstraction which is different from the trivial abstraction $A^{\top}$: this is a two-point abstraction $\{\mathbb{Z}, K\}$ for some set of integers $K \in \wp(\mathbb{Z})$. To be concrete and simple, let $K = \{0\}$, so that we consider the corresponding abstraction $A^0 \triangleq \{\mathbb{Z}, \{0\}\} \in \mathrm{Abs}(\wp(\mathbb{Z}))$. Note that A^0 is only able to represent precisely the fact that an integer variable has a value equals to 0 and that $A^0(\varnothing) = \{0\}$. Let us observe that A^0 is not complete for the transfer function $(\!|\, x > 0 \,|\!)$: for example, we have that $A^0\big((\!|\, x > 0 \,|\!)\{-1\}\big) = A^0(\varnothing) = \{0\} \subsetneq \mathbb{Z} = (\!|\, x > 0 \,|\!)^{A^0} A^0(\{-1\})$. Why A^0 is not complete for the test $x > 0$? As the example highlights, the problem lies in the fact that if S is a set of integers that do not satisfy the test $x > 0$ then the abstraction of $(\!|\, x > 0 \,|\!)S$ boils down to the abstraction of the empty set $\varnothing$ which is $\{0\}$, whereas the only option for A^0 is to abstract one such set S to $\mathbb{Z}$ and this gives rise to incompleteness. Suitable refinements of the abstraction A^0 can avoid this incompleteness: one such refined abstraction should be complete for $(\!|\, x > 0 \,|\!)$ while preserving the expressiveness of A^0. For example, the Sign abstraction [8] depicted below on the left



turns out to be complete for $(\!|\, x > 0 \,|\!)$ and, clearly, is more precise than A^0. Yet, for the specific goal of being complete for $(\!|\, x > 0 \,|\!)$, Sign is too refined, i.e. precise. In fact, the abstraction A' depicted above in the middle is still complete for $(\!|\, x > 0 \,|\!)$, while being simpler than Sign and more precise than A^0. What we really need here is a *minimal refinement* of A^0 which is complete for $(\!|\, x > 0 \,|\!)$. This type of abstraction refinements have been called *complete shell* refinements [13].

3.1 Shells

Recall that abstractions of a common concrete domain D are preordered w.r.t. their relative precision: If $A_1, A_2 \in \mathrm{Abs}(D)$, where $\langle \alpha_i, \gamma_i \rangle$ are the corresponding Galois connections, then $A_1 \trianglelefteq A_2$, i.e. A_1 is a refinement of A_2 and A_2 is a simplification of A_1, when for all $d \in D$, $\gamma_1(\alpha_1(d)) \leq_D \gamma_2(\alpha_2(d))$. Moreover, A_1 and A_2 are equivalent when $A_1 \trianglelefteq A_2$ and $A_2 \trianglelefteq A_1$. By a slight abuse of notation, $\mathrm{Abs}(D)$ denotes the family of abstractions of D up to the above equivalence. It is well known [9] that $\langle \mathrm{Abs}(D), \trianglelefteq \rangle$ is a complete lattice. Given a family of abstract domains $\mathcal{X} \subseteq \mathrm{Abs}(D)$, their lub $\sqcup\mathcal{X}$ is therefore the most precise domain in $\mathrm{Abs}(D)$ which is a simplification of any domain in $\mathcal{X}$.

In a general abstract interpretation framework, Giacobazzi et al. [13] have shown that an abstraction A can be made complete for a family of continuous concrete functions through a minimal refinement of A. Consider a continuous concrete function $f : D \to D$ and an abstraction $A \in \mathrm{Abs}(D)$. Then, A is refined to the complete shell $\mathrm{CShell}_f(A)$ which can be obtained as follows (we use a simplified characterization proved in [2, Theorem 4.1]). We first define a transform $R_f : \wp(D) \to \wp(D)$

as $R_f(X) \triangleq \cup_{a \in X} \max\{d \in D \mid f(d) \leq a\}$, namely $R_f(X)$ collects, for any value $a \in X$, the maximal concrete values d whose f-image is approximated by a. Then, we define $\mathrm{CShell}_f(A) \triangleq \mathrm{Cl}_\wedge(R_f^\omega(\gamma(A)))$, where $\mathrm{Cl}_\wedge$ denotes the closure under glb's $\wedge$ of subsets of D and R_f^ω denotes the ω-closure of R_f, i.e., $R_f^\omega(X) \triangleq \cup_{i \in \mathbb{N}} R_f^i(X)$ (note that $R_f^0(X) = X$). The main result in [13] shows that

$$\mathrm{CShell}_f(A) = \sqcup\{A' \in \mathrm{Abs}(D) \mid A' \trianglelefteq A,\ A' \text{ is complete for } f\}$$

namely, $\mathrm{CShell}_f(A)$ is the least f-complete refinement of A.

Similarly, an abstraction can be also minimally refined in order to be exact for any given set of functions [12]. Given a concrete function $f : D \to D$, any abstraction $A \in \mathrm{Abs}(D)$ can be refined to the exact shell $\mathrm{EShell}_f(A)$ for f as follows: we define $S_f : \wp(D) \to \wp(D)$ as $S_f(X) \triangleq \{f(x) \in D \mid x \in X\}$ and $\mathrm{EShell}_f(A) \triangleq \mathrm{Cl}_\wedge(S_f^\omega(\gamma(A)))$. Thus, $S_f^\omega(\gamma(A))$ is the closure of the abstraction A (more precisely, of its concrete image $\gamma(A)$) under applications of the concrete function f while the exact shell $\mathrm{EShell}_f(A)$ is the closure of $S_f^\omega(\gamma(A))$ under glb's in D. Here, we have that

$$\mathrm{EShell}_f(A) = \sqcup\{A' \in \mathrm{Abs}(D) \mid A' \trianglelefteq A,\ A' \text{ is exact for } f\}.$$

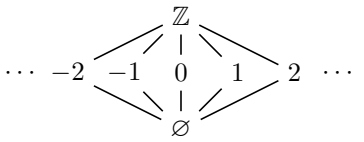
3.2 Signs

The above general solution [13] tells us that in order to make A complete we need to iteratively add to A the maximal concrete values d such that $f(d) \leq a$ for some abstract value $a \in A$ until this process of adding new values to A converges, and at the end this set of concrete values must be closed under glb's. Thus, for the abstract value $\{0\} \in A^{\mathrm{o}}$, we need to consider $\max\{S \in \wp(\mathbb{Z}) \mid (\!|x > 0|\!)S \subseteq \{0\}\}$ and this provides the concrete value $\mathbb{Z}_{\leq 0}$ to join to A^{o}. In turn, $\max\{S \in \wp(\mathbb{Z}) \mid (\!|x > 0|\!)S \subseteq \mathbb{Z}_{\leq 0}\} = \mathbb{Z}_{\leq 0}$, so that we do not need to further refine A^{o}. Hence, $\mathrm{Shell}_{(\!|x>0|\!)}(A) = A'$. Likewise, we have that $A'' = \{\mathbb{Z}, \mathbb{Z}_{\geq 0}, 0\}$ (which is depicted above) is the complete shell refinement of A for the transfer function $(\!|x < 0|\!)$. Furthermore, Sign can be obtained as complete shell refinement of A for both transfer functions $(\!|x > 0|\!)$ and $(\!|x < 0|\!)$.

We have therefore analyzed the genesis of a toy abstract domain like Sign from the viewpoint of completeness: Sign is characterized as the minimal refinement of a basic domain $\{\mathbb{Z}, 0\}$ whose abstract transfer functions for the boolean tests $x > 0?$ and $x < 0?$ are complete. In the following, we will show that this completeness-based view on abstraction design can be very useful to understand the genesis of a range of numerical abstractions.

3.3 Constant Propagation

Let us consider the standard abstraction CP used in constant propagation analysis [16]:



CP is more precise than the basic abstraction A^0, hence it makes sense to ask whether CP can be viewed as a complete shell refinement of A^0 for some transfer functions. From the viewpoint of its completeness properties, one may observe that CP is clearly complete for the transfer functions $(\!|x := x + k|\!)$, for any $k \in \mathbb{Z}$. Moreover, we also notice that if an abstraction A is complete for $(\!|x := x + 1|\!)$ and $(\!|x := x - 1|\!)$ then, since completeness is preserved by composing functions, A is also complete for all the transfer functions $(\!|x := x + k|\!)$, for all $k \in \mathbb{Z}$, and this allows us to focus on $(\!|x := x \pm 1|\!)$ only. Actually, it turns out that completeness for $(\!|x := x \pm 1|\!)$ completely characterizes constant propagation CP, in the sense that the complete shell of A^0 for both $(\!|x := x + 1|\!)$ and $(\!|x := x - 1|\!)$ is precisely CP. To this aim, it is enough to note that

$$\begin{array}{ll} \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x - 1|\!)S \subseteq \{0\}\} = \{1\} & \\ \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x - 1|\!)S \subseteq \{1\}\} = \{2\} & \cdots \\ \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x + 1|\!)S \subseteq \{0\}\} = \{-1\} & \\ \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x + 1|\!)S \subseteq \{-1\}\} = \{-2\} & \cdots \end{array}$$

and that at the end the closure under intersection adds the empty set $\varnothing$.

3.4 Intervals

Sign is not complete for the transfer function $(\!|x := x + 1|\!)$. In fact, notice that

$$\begin{array}{l} \mathrm{Sign}((\!|x := x + 1|\!)\{-1\}) = \mathrm{Sign}(\{0\}) = \{0\} \\ (\!|x := x + 1|\!)^{\mathrm{Sign}}\,\mathrm{Sign}(\{-1\}) = (\!|x := x + 1|\!)^{\mathrm{Sign}}(\mathbb{Z}_{\leq 0}) = \mathbb{Z} \end{array}$$

The well-known domain Int of integer intervals [8] is instead clearly complete for $(\!|x := x + 1|\!)$ and $(\!|x := x - 1|\!)$. As shown in [13], the complete shell of Sign for $(\!|x := x + 1|\!)$ and $(\!|x := x - 1|\!)$ turns out to be Int. In fact, we have that:

$$\begin{array}{ll} \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x \pm 1|\!)S \subseteq \mathbb{Z}_{\leq 0}\} = \mathbb{Z}_{\leq \mp 1} & \\ \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x \pm 1|\!)S \subseteq \mathbb{Z}_{\leq -1}\} = \mathbb{Z}_{\leq \mp 2} & \cdots \\ \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x \pm 1|\!)S \subseteq \mathbb{Z}_{\geq 0}\} = \mathbb{Z}_{\geq \mp 1} & \\ \max\{S \in \wp(\mathbb{Z}) \mid (\!|x := x \pm 1|\!)S \subseteq \mathbb{Z}_{\geq 1}\} = \mathbb{Z}_{\geq \mp 2} & \cdots \end{array}$$

and clearly the family of sets $\mathbb{Z}_{\leq k}$ and $\mathbb{Z}_{\geq k}$, where k ranges in $\mathbb{Z}$, generate by intersection all the integer intervals in Int.

It is worth noting that, in general, completeness is not preserved under abstraction refinements. In fact, while Sign is complete for $(\!|x > 0|\!)$, its complete shell Int for $(\!|x := x \pm 1|\!)$ loses this completeness: for example, we have that $\mathrm{Int}((\!|x > 0|\!)\{0, 2\}) = [2, 2]$ while $(\!|x > 0|\!)^{\mathrm{Int}}\,\mathrm{Int}(\{0, 2\}) = [1, +\infty) \sqcap [0, 2] = [1, 2]$. It is simple to observe that if we try to compensate this lack by the complete shell of Int for $(\!|x > k|\!)$, for all $k \in \mathbb{Z}$, we end up with the whole concrete domain $\wp(\mathbb{Z})$: this depends on the fact that

$$\max\{S \in \wp(\mathbb{Z}) \mid (\!|x > k|\!)S \subseteq \mathbb{Z}_{\geq k+2}\} = \mathbb{Z}_{\neq k+1}$$

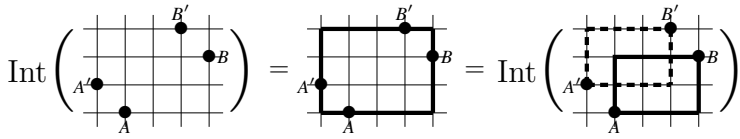
and then by closing under intersections the family of sets $\{\mathbb{Z}_{\neq k}\}_{k \in \mathbb{Z}}$ we get $\wp(\mathbb{Z})$. If instead we consider the complete shell of Int for the specific transfer function $(\!|x > 0|\!)$ then Int would be refined to $\mathrm{Int}^{\neq 1} \triangleq \mathrm{Int} \cup \{I \smallsetminus \{1\} \mid I \in \mathrm{Int}\}$, namely $\mathrm{Int}^{\neq 1}$ is able to represent precisely integer intervals with an inner hole $\{1\}$.

3.5 Octagons

Let us consider the program $P \equiv x := 0;\ y := 4;\ \textbf{while}\, x \neq 0\, \textbf{do}\, \{x{-}{-};\ y{+}{+}\}$. According to Miné [17], this is one paradigmatic program showing the need for relational abstract domains. In fact, we have that the interval abstraction Int is not complete for P. Here, $\mathrm{Store} = \wp(\mathbb{Z}^2)$ so that:

$$\mathrm{Int}([\![P]\!]\langle x/\{4\}, y/\{0\}\rangle) = \langle x/[0,0], y/[4,4]\rangle$$
$$[\![P]\!]^{\mathrm{Int}}\,\mathrm{Int}(\langle x/\{4\}, y/\{0\}\rangle) = \langle x/[0,0], y/[0,+\infty)\rangle$$

Of course, the problem of Int with P is that Int is not able to represent precisely the invariant property $x + y = 4$ of the while-loop. Miné [17] then proposes to refine intervals to the so-called octagon abstraction $\mathrm{Oct} \in \mathrm{Abs}(\wp(\mathrm{Store}))$, which is able to express precisely sets of integers like $\{\langle x, y\rangle \in \mathbb{Z}^2 \mid x \pm y = k\}$, where $k \in \mathbb{Z}$. Observe that a set like $S = \{\langle x/0, y/4\rangle, \langle x/2, y/2\rangle, \langle x/4, y/0\rangle\}$ cannot be represented precisely in Oct, since Oct must add $\langle x/1, y/3\rangle$ and $\langle x/3, y/1\rangle$ in the approximation of S. Thus, the observation here is that Oct is exact for the function $\lambda S.S \cup (\!|\, x{-}{-};\ y{+}{+}\,|\!)S$ which is iterated in the fixpoint computation of the while-loop of P. It is worth remarking that Oct is also complete for $\lambda S.S \cup (\!|\, x{-}{-};\ y{+}{+}\,|\!)S$, but this is not the distinguishing feature of Oct compared to Int because Int is already complete for this function, as the following picture suggests:



What we really need is an abstract domain which is able to express precisely the invariant $x + y = 4$ in each iteration step in the fixed point computation of the while-loop of P. We thus refine Int to its exact shell for the function $F \triangleq \lambda S.S \cup (\!|\, x{-}{-};\ y{+}{+}\,|\!)S$ and, dually, for $G \triangleq \lambda S.S \cup (\!|\, x{+}{+};\ y{-}{-}\,|\!)S$. This exact shell can be simply characterized as follows: for a generic point $\langle a, b\rangle \in \wp(\mathbb{Z}^2)$, we have that

$$F^0(\{\langle a, b\rangle\}) = \{\langle a, b\rangle\}$$
$$F^1(\{\langle a, b\rangle\}) = \{\langle a, b\rangle, \langle a-1, b+1\rangle\}$$
$$F^2(\{\langle a, b\rangle\}) = \{\langle a, b\rangle, \langle a-1, b+1\rangle, \langle a-2, b+2\rangle\}$$
$$\ldots$$

so that $F^\omega(\{\langle a, b\rangle\}) = \{\langle x, y\rangle \in \mathbb{Z}^2 \mid x + y = a + b,\ x \leq a,\ y \geq b\}$. Likewise, we have that $G^\omega(\{\langle a, b\rangle\}) = \{\langle x, y\rangle \in \mathbb{Z}^2 \mid x - y = a + b,\ x \geq a,\ y \leq b\}$. Then, by closing under intersections Int, $\{F^\omega(\{\langle a, b\rangle\})\}_{\langle a,b\rangle \in \mathbb{Z}^2}$ and $\{G^\omega(\{\langle a, b\rangle\})\}_{\langle a,b\rangle \in \mathbb{Z}^2}$ we precisely get all the octagons in Oct.

3.6 Polyhedra

The well-known polyhedra abstraction [10] can be characterized as a simple further step of exact refinement of octagons. It is enough to consider a generic program scheme like

$$Q \equiv x := x_0;\, y := y_0;\, \textbf{while}\,(*)\,\textbf{do}\,\{x := x + k_x;\ y := y + k_y\}$$

Of course, in this case Oct fails to represent precisely the invariant $k_x y - k_y x = k_x y_0 - k_y x_0$ of the while-loop of Q.

It is therefore clear that the exact shell refinement of Oct (or Int) for the family of functions $F = \{\lambda S.S \cup (\!| x := x + k_x;\ y := y + k_y |\!) S\}_{\langle k_x, k_y\rangle \in \mathbb{Z}^2}$ adds to Oct all the linear sets $\{\langle x, y\rangle \in \mathbb{Z}^2 \mid ax + by = c\}$, where a, b, c range in $\mathbb{Z}$, so that the closure under intersections (i.e., logical conjunction) precisely provides the polyhedra abstraction.

4 Completeness in Model Checking

Consider a system model specified as a Kripke structure $\mathcal{K} = (\mathrm{State}, \rightarrow)$. Standard abstract model checking [5] consists in approximating $\mathcal{K}$ by an abstract Kripke structure $\mathcal{A} = (\mathrm{State}^\sharp, \rightarrow^\sharp)$, where the set $\mathrm{State}^\sharp$ of abstract states is defined by a surjective map $h : \mathrm{State} \rightarrow \mathrm{State}^\sharp$ that joins together indistinguishable concrete states. Thus, $\mathrm{State}^\sharp$ determines a partition of State and vice versa any partition of State can be viewed as a set of abstract states. It is simple to view state partitions as abstractions [19]. Any state partition $P \in \mathrm{Part}(\mathrm{State})$ can be viewed as an abstraction $P^a \in \mathrm{Abs}(\wp(\mathrm{State})_\subseteq)$ that approximates any set S of states by the minimal cover of S in the partition P, i.e., $\cup\{B \in P \mid B \cap S \neq \varnothing\}$. Hence, $P^a \triangleq \{S \subseteq \mathrm{State} \mid \exists\{B_i\}_{i\in I} \subseteq P.\ S = \cup_{i\in I} B_i\}$, ordered by subset inclusion, is viewed as an abstraction in $\mathrm{Abs}(\wp(\mathrm{State}))$ that encodes the state partition P. Furthermore, if $A \in \mathrm{Abs}(\wp(\mathrm{State}))$ is any abstraction then A is the encoding of some state partition P, i.e. $A = P^a$, iff A is exact for the negation function $\neg : \wp(\mathrm{State}) \rightarrow \wp(\mathrm{State})$ such that $\neg S = \mathrm{State} \smallsetminus S$. This embedding of the lattice of partitions $\mathrm{Part}(\mathrm{State})$ into the lattice of abstractions $\mathrm{Abs}(\wp(\mathrm{State}))$ allows us to reason on the completeness and exactness properties of partitions, and hence of abstract Kripke structures.

4.1 Strong Preservation

Given some temporal specification language $\mathfrak{L}$ — like CTL, ACTL, μ-calculus, etc. — and a corresponding interpretation of its formulae on the states of a Kripke structure $\mathcal{K}$, an abstract Kripke structure $\mathcal{A}$ *preserves* $\mathfrak{L}$ when for any $\varphi \in \mathfrak{L}$ and $s \in \mathrm{State}$, $h(s) \models^{\mathcal{A}} \varphi \Rightarrow s \models^{\mathcal{K}} \varphi$, while $\mathcal{A}$ *strongly preserves* $\mathfrak{L}$ when $h(s) \models^{\mathcal{A}} \varphi \Leftrightarrow s \models^{\mathcal{K}} \varphi$ holds.

Given a language $\mathfrak{L}$ and a Kripke structure $\mathcal{K}$, a well-known key problem is to compute the smallest abstract state space $\mathrm{State}^\sharp_{\mathfrak{L}}$, when this exists, such that one can define an abstract Kripke structure $\mathcal{A}_{\mathfrak{L}} = (\mathrm{State}^\sharp_{\mathfrak{L}}, \rightarrow^\sharp)$ that strongly preserves $\mathfrak{L}$. This problem admits solution for a number of well-known temporal languages like Hennessy-Milner logic HML, CTL, ACTL and CTL-X (i.e. CTL without the next-time operator X). A number of algorithms for solving this problem exist, like those by Paige and Tarjan [18] for HML and CTL, by Henzinger et al. [15] and Ranzato and Tapparo [21] for ACTL, and Groote and Vaandrager [14] for CTL-X. These are *coarsest partition refinement* algorithms: given a language $\mathfrak{L}$ and an initial state partition P, which is determined by a state labeling in $\mathcal{K}$, these algorithms can be viewed as computing the coarsest partition $P_{\mathfrak{L}}$ that refines P and induce an abstract Kripke structure that strongly

preserves $\mathfrak{L}$. It is worth remarking that these algorithms have been designed for computing some well-known behavioural equivalences used in process algebra like bisimulation (for CTL), simulation (for ACTL) and stuttering bisimulation (for CTL-X). Our abstract interpretation approach allows us to provide a generalized view of these partition refinement algorithms based on exactness properties.

4.2 Bisimulation

Given a partition $P \in \mathrm{Part}(\mathrm{State})$ and a state s, $P(s)$ denotes the block of P that contains s. Then, P is a bisimulation on a Kripke structure $\mathcal{K}$ when $P(s) = P(t)$ and $s \to s'$ imply that there exists some state t' such that $t \to t'$ and $P(s') = P(t')$. It is well known [5] that P is a bisimulation iff the abstract Kripke structure $\mathcal{A}_P = \langle P, \to^{\exists} \rangle$ strongly preserves HML (or, equivalently, CTL), where $B_1 \to^{\exists} B_2$ iff there exist $s_i \in B_i$ such that $s_1 \to s_2$. Moreover, the coarsest partition in $\mathrm{Part}(\mathrm{State})$ which is a bisimulation on $\mathcal{K}$ exists and is called bisimulation equivalence.

Bisimulation for P can be expressed as an exacteness property of the abstraction P^a [19]. The standard predecessor operator $\mathrm{pre} : \wp(\mathrm{State}) \to \wp(\mathrm{State})$ on $\mathcal{K}$ is defined as $\mathrm{pre}(T) \triangleq \{s \in \mathrm{State} \mid s \to t,\ t \in T\}$ and is here considered as a concrete function. Hence, it turns out that P is a bisimulation iff P^a is exact for pre. As a consequence, any partition refinement algorithm for computing bisimulation can be characterized as an exact shell abstraction refinement for: (1) the negation operator $\neg$, in order to ensure that the abstraction is indeed a partition and (2) the predecessor operator pre, in order to ensure that the partition is a bisimulation. In particular, the Paige-Tarjan bisimulation algorithm [18] can be viewed as an efficient implementation of this exact shell computation.

4.3 Simulation

Given a preorder relation $R \in \mathrm{PreOrd}(\mathrm{State})$ on states, $R(s)$ denotes $\{t \in \mathrm{State} \mid (s,t) \in R\}$ while $P_R \triangleq R \cap R^{-1} \in \mathrm{Part}(\mathrm{State})$ denotes the symmetric reduction of R, which being an equivalence relation can be viewed as a state partition.

A preorder R is a simulation on a Kripke structure $\mathcal{K}$ if $t \in R(s)$ and $s \to s'$ imply that there exists some state t' such that $t \to t'$ and $t' \in R(s')$. It is well known [5] that a preorder R is a simulation iff the abstract Kripke structure $\mathcal{A}_R = \langle P_R, \to^{\exists} \rangle$ strongly preserves ACTL and that the largest preorder in $\mathrm{PreOrd}(\mathrm{State})$ which is a simulation on $\mathcal{K}$ exists and is called simulation preorder.

The characterization of simulation as an exactness property follows the lines of bisimulation [19]. Similarly to partitions, a preorder relation R can be viewed as an abstraction $R^a \in \mathrm{Abs}(\wp(\mathrm{State}))$ that approximates any set $S \subseteq \mathrm{State}$ with the image of S through R, i.e. $\cup_{s \in S} R(s)$. Thus, $R^a \triangleq \{S \subseteq \mathrm{State} \mid \exists\, S \subseteq \mathrm{State}\,.\, S = \cup_{s \in S} R(s)\}$, ordered by subset inclusion, is viewed as an abstraction in $\mathrm{Abs}(\wp(\mathrm{State}))$ that encodes the preorder relation R. On the other hand, an abstraction $A \in \mathrm{Abs}(\wp(\mathrm{State}))$ is the encoding of some state preorder R, i.e. $A = R^a$, iff A is exact for the union $\cup : \wp(\mathrm{State})^2 \to \wp(\mathrm{State})$. We notice that A is exact for the union iff A is a so-called disjunctive abstraction, so that the lattice of preorder relations $\mathrm{PreOrd}(\mathrm{State})$ can be embedded into $\mathrm{Abs}(\wp(\mathrm{State}))$ as the sublattice of disjunctive abstractions. This allows us to get the following characterization: a preorder R is a simulation iff R^a is exact for

pre. Here, we obtain that any algorithm for computing the simulation preorder can be characterized as an exact shell abstraction refinement for: (1) the union $\cup$, in order to ensure that the abstraction represents a preorder and (2) the predecessor operator pre, in order to ensure that this preorder is a simulation. In particular, the simulation algorithm with the best time complexity [21] has been designed as an efficient implementation of this exact shell computation.

4.4 Stuttering Bisimulation and Simulation

Stuttering bisimulation and simulation relations characterize temporal logics which do not feature the next-time connective X. Let us focus on stuttering simulation. A preorder relation $R \in \mathrm{PreOrd}(\Sigma)$ is a stuttering simulation on $\mathcal{K}$ if $t \in R(s)$ and $s \to s'$ imply that there exist states $t_0, ..., t_k$, with $k \geq 0$, such that: (i) $t_0 = t$, (ii) for all $i \in [0, k[$, $t_i \to t_{i+1}$ and $t_i \in R(s)$, (iii) $t_k \in R(s)$. It turns out that stuttering simulation can be characterized as an exactness property [19]. Following [14], consider the binary stuttering operator $\mathrm{pos} : \wp(\mathrm{State}) \times \wp(\mathrm{State}) \to \wp(\mathrm{State})$ which is defined as follows:

$$\mathrm{pos}(S, T) \triangleq$$
$$\{s \in S \mid \exists k \geq 0. \exists s_0, ..., s_k.\ s_0 = s\ \&\ \forall i \in [0, k).\ s_i \in S,\ s_i \to s_{i+1}\ \&\ s_k \in T\}.$$

Hence, it turns out that a preorder R is a stuttering simulation iff R^a is exact for pos. As shown in [20], this allows us to design an efficient algorithm for computing the stuttering simulation preorder as an exact shell abstraction refinement for the union $\cup$ and the stuttering operator pos.

4.5 Probabilistic Bisimulation and Simulation

The main behavioral relations between concurrent systems, like simulation and bisimulation, have been studied in probabilistic extensions of reactive systems like Markov chains and probabilistic automata. As recently shown in [11], we mention that bisimulation and simulation relations on probabilistic transition systems can still be characterized as exactness properties in abstract interpretation and as a byproduct this allows to design efficient algorithms that compute these behavioral relations as exact shell refinements.

5 Conclusion

We have shown how completeness and exactness properties of abstractions play an ubiquitous role in static analysis by exhibiting an array of examples ranging from abstract interpretation-based program analysis to abstract model checking. We are convinced that it is often rewarding to look at scenarios based on some form of semantic approximation under the light of completeness/exactness: this can be useful both to understand the deep logic of the approximation and to profit of the beneficial toolkit that completeness brings, like complete and exact shell refinements.

Acknowledgements. I would like to thank Roberto Giacobazzi for his constant and passionate brainstorming on completeness and exactness: Roberto and I are both persuaded that completeness can be spotted really everywhere.

References

1. Aarts, C., Backhouse, R., Boiten, E., Doornbos, H., van Gasteren, N., van Geldrop, R., Hoogendijk, P., Voermans, E., van der Woude, J.: Fixed-point calculus. Inform. Process. Lett. 53(3), 131–136 (1995)
2. Amato, G., Scozzari, F.: Observational completeness on abstract interpretation. Fundamenta Informaticae 47(12), 1533–1560 (2011)
3. Apt, K.R., Plotkin, G.D.: Countable nondeterminism and random assignment. J. ACM 33(4), 724–767 (1986)
4. de Bakker, J.W., Meyer, J.-J.C., Zucker, J.: On infinite computations in denotational semantics. Theoret. Comput. Sci. 26(1-2), 53–82 (1983)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. The MIT Press (1999)
6. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. Ph.D. dissertation, Université Scientifique et Médicale de Grenoble, Grenoble, France (1978)
7. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277(1-2), 47–103 (2002)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM POPL, pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. 6th ACM POPL, pp. 269–282 (1979)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. 5th ACM POPL, pp. 84–97 (1978)
11. Crafa, S., Ranzato, F.: Bisimulation and simulation algorithms on probabilistic transition systems by abstract interpretation. Formal Methods in System Design 40(3), 356–376 (2012)
12. Giacobazzi, R., Quintarelli, E.: Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 356–373. Springer, Heidelberg (2001)
13. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM 47(2), 361–416 (2000)
14. Groote, J.F., Vaandrager, F.: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
15. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. 36th FOCS, pp. 453–462. IEEE Press (1995)
16. Kildall, G.: A unified approach to global program optimization. In: Proc. 1st ACM POPL, pp. 194–206 (1973)
17. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
18. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
19. Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. Logic and Computation 17(1), 157–197 (2007)
20. Ranzato, F., Tapparo, F.: Computing Stuttering Simulations. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 542–556. Springer, Heidelberg (2009)
21. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Information and Computation 208(1), 1–22 (2010)

Abstraction-Guided Synthesis*

Eran Yahav

Technion - Israel Institute of Technology
yahave@cs.technion.ac.il

Abstract. Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When $P \not\models_\alpha S$, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In contrast, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors — to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this talk, we focus on *concurrent programs*, and consider changes to P that correspond to restricting program executions by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can then be avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

We show how abstraction-guided synthesis can be applied to automatically synthesize atomic sections, conditional critical regions, inter-thread ordering constraints, and memory fences.

This talk is based on joint work with Martin Vechev, Greta Yorsh, Michael Kuperstein, and Veselin Raychev.

* This work was partially supported by The Israeli Science Foundation (grant no. 965/10).

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, p. 27, 2013.

SMT-Based Bisimulation Minimisation of Markov Models

Christian Dehnert[1], Joost-Pieter Katoen[1], and David Parker[2]

[1] RWTH Aachen University, Germany
[2] University of Birmingham, United Kingdom

Abstract. Probabilistic model checking is an increasingly widely used formal verification technique. However, its dependence on computationally expensive numerical operations makes it particularly susceptible to the state-space explosion problem. Among other abstraction techniques, bisimulation minimisation has proven to shorten computation times significantly, but, usually, the full state space needs to be built prior to minimisation. We present a novel approach that leverages satisfiability solvers to extract the minimised system from a high-level description directly. A prototypical implementation in the framework of the probabilistic model checker Prism provides encouraging experimental results.

1 Introduction

Markov chains are omnipresent. They are used in reliability analysis, randomised algorithms and performance evaluation. In the last decade, probabilistic model checking has emerged as a viable and efficient alternative to classical analysis techniques for Markov chains, which typically focus on transient and long-run probabilities. This growing popularity is mainly due to the availability of ever improving software tools such as Prism [15] and Mrmc [11]. Like traditional model checkers, these tools suffer from the curse of dimensionality—the state space grows exponentially in the number of system components and variables. As numerical computations are at the heart of verifying Markov chains, several numerical values need to be stored for each (reachable) state in addition to the model itself, making the state space explosion problem even more pressing.

A variety of techniques has been recently developed to reduce the state space of probabilistic models prior to verification. These include symmetry reduction [14], bisimulation minimisation [10] and abstraction, e.g., in the abstraction-refinement paradigm [13,9]. Bisimulation [16] (also known as ordinary lumping [3]) is of particular interest as it preserves widely used logics such as PCTL*, PCTL and probabilistic LTL [1], and its coarsest quotient can be efficiently computed [5,18]. In contrast to traditional model checking [6], it has proven to significantly shorten computation times [10]. The main drawback of bisimulation quotienting algorithms, however, is that they typically require the entire reachable state space. Techniques to alleviate this effect include the use of data structures based on binary decision diagrams (BDDs) to reduce storage

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 28–47, 2013.

costs [21,20,17], and compositional minimisation techniques [4]. This paper takes a radically different approach: we extract the bisimulation quotient directly from a high-level description using SMT (satisfiability modulo theories) solvers.

The starting-point of our approach is to take a probabilistic program representing a Markov chain, described in the PRISM modelling language. This probabilistic program consists of guarded commands, each of which includes a probabilistic choice over a set of assignments that manipulate program variables. The main idea is to apply a partition-refinement bisimulation quotienting in a truly symbolic way. Each block in the partition is represented by a Boolean expression and weakest preconditions of guarded commands yield the refinement of blocks. All these computation steps can be dispatched as standard queries to an SMT solver. The quotient distribution, over blocks, for a state is obtained by summing up the probabilities (given by a command) attached to assignments that lead into the respective blocks. This is determined using an ALLSAT enumeration query by the SMT solver. Whereas similar techniques have been used for obtaining over-approximations of Markov models [19,12] —yielding sound abstractions for a "safe" fragment of PCTL properties— this is (to the best of our knowledge) the first bisimulation quotienting approach based on satisfiability solvers. This paper focuses on bisimulation for discrete-time Markov chains (DTMCs), but the techniques are also directly applicable to continuous-time Markov chains (CTMCs). In addition, in Section 4, we discuss how to extend them to models that incorporate nondeterminism, such as MDPs or CTMDPs.

Experiments on probabilistic verification benchmarks show encouraging results: speed-ups in total verification time by up to two orders of magnitude over PRISM on one example, and, on another, scalability to models that go far beyond the capacity of current verification engines such as PRISM and MRMC. A comparison with the symbolic bisimulation engine SIGREF [21] also yields favourable results. Although —like for BDD-based quotienting— there are examples with smaller improvements or where this approach is inferior, we believe that SMT-based quotienting offers clear potential. Other advantages of our approach are that it is applicable to infinite probabilistic programs whose bisimulation quotient is finite and that it is directly applicable to parametric Markov chains [8].

Related Work. For non-probabilistic models, Graf and Saïdi [7] pioneered *predicate abstraction*, which uses Boolean predicates and weakest precondition operations to compute abstractions. This technique is rather en vogue in software model checking, as is the use of SAT/SMT solvers to build abstractions [2]. Predicate abstraction has been adapted to the probabilistic setting [19,12] and used to obtain abstractions of probabilistic models, e.g., through CEGAR [9] or game-based abstraction refinement [13]. However, these methods usually only focus on a specific subset of PCTL* and do not compute precise abstractions but over-approximations. While the latter can be beneficial in some cases, in general it requires a separate abstraction for each property to be verified. In contrast, bisimulation minimisation is a precise abstraction, which coincides with PCTL* equivalence [1], so all properties that can be formulated in that logic are preserved. Another important difference is that probabilistic abstraction-refinement

techniques [9,13] usually require an expensive numerical solution phase to perform refinement, whereas partition refinement for bisimulation is comparatively cheap. Omitting numerical analysis also makes our approach easily extendable to systems with continuous time and parameters. Wimmer et al. [21] use BDDs to symbolically compute the bisimulation quotient. However, this requires the construction of a BDD representation of the state space of the full, unreduced model prior to minimisation, which we deliberately try to avoid.

2 Preliminaries

We begin with some brief background material on discrete-time Markov chains, bisimulation minimisation and the PRISM modelling language.

Markov Models. Markov models are similar to transition systems in that they comprise states and transitions between these states. In discrete-time Markov chains, each state is equipped with a discrete probability distribution over successor states. Let $Dist_S$ be the set of discrete probability distributions over S.

Definition 1 (Discrete-time Markov chain (DTMC)). *A DTMC is a tuple* $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$ *where (i)* S *is a countable, non-empty set of states, (ii)* $\mathbf{P} : S \times S \to [0,1]$ *is the transition probability function that assigns to each pair* (s, s') *of states the probability* $\mathbf{P}(s, s')$ *of moving from state* s *to* s' *in one step such that* $\mathbf{P}(s, \cdot) \in Dist_S$*, (iii)* $s_{init} \in S$ *is the initial state, (iv)* AP *is a set of atomic propositions, and (v)* $L : S \to 2^{AP}$ *is the labelling function that assigns a (possibly empty) set of atomic propositions* $L(s)$ *to a state* $s \in S$*.*

For quantitative reasoning over DTMCs, we focus on the logic PCTL*, a probabilistic extension of CTL* which subsumes other common logics such as PCTL and probabilistic LTL. Instead of the path quantifiers (A and E) of CTL*, it features the $\mathcal{P}_{\bowtie p}(\varphi)$ operator, which asserts that the probability mass of paths satisfying φ satisfies $\bowtie p$, where $\bowtie \in \{<, \leq, >, \geq\}$. For example, the property "the probability that the system eventually fails lies below 70%" can be expressed as $\mathcal{P}_{<0.7}(\Diamond \mathit{fail})$. Let $s \models_{\mathcal{D}} \Phi$ denote that state s of a DTMC $\mathcal{D}$ satisfies a formula Φ. If $s_{init} \models_{\mathcal{D}} \Phi$, the DTMC $\mathcal{D}$ satisfies Φ and we denote this by $\mathcal{D} \models \Phi$.

Bisimulation minimisation. Given a DTMC $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$ and an equivalence relation R on S, the set of equivalence classes of S under R is denoted S/R and $[s]_R$ is the equivalence class of $s \in S$ under R.

Definition 2 (Quotient distribution). *For equivalence relation* R *and distribution* $\mu \in Dist_S$*, the* quotient distribution *of* μ *with respect to* R*, denoted* $[\mu]_R \in Dist_{S/R}$*, is defined by* $[\mu]_R(C) = \sum_{s \in C} \mu(s)$ *for all* $C \in S/R$*.*

Definition 3 ((Strong) probabilistic bisimulation). *An equivalence relation* R *on* S *is a* (strong) bisimulation *on* $\mathcal{D}$ *if, for all* $(s_1, s_2) \in R$*:*

$$L(s_1) = L(s_2) \text{ and } \mathbf{P}(s_1, C) = \mathbf{P}(s_2, C) \text{ for all } C \in S/R$$

where $\mathbf{P}(s, C)$ *denotes the sum* $\sum_{s' \in C} \mathbf{P}(s, s')$*.*

States $s_1, s_2 \in S$ are (strongly) bisimilar, denoted $s_1 \sim s_2$, if there exists a bisimulation on $\mathcal{D}$ that relates s_1 and s_2. Note that $\sim$ is the coarsest bisimulation on $\mathcal{D}$. Intuitively, bisimilar states can stepwise simulate each other, meaning that they can be merged while preserving important properties.

Definition 4 (Bisimulation quotient). *For a DTMC* $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$, *the* bisimulation quotient *is defined as* $\mathcal{D}/\!\sim = (S/\!\sim, \mathbf{P}', [s_{init}]_\sim, AP, L')$, *where*

$$\mathbf{P}'([s]_\sim, [t]_\sim) = \mathbf{P}(s, [t]_\sim) \text{ and } L'([s]_\sim) = L(s).$$

Note that $\mathcal{D}/\!\sim$ is well-defined. The preservation theorem of Aziz et al. [1] states that strong bisimulation is sound and complete with respect to PCTL*:

Proposition 1. $s \models_{\mathcal{D}} \Phi \iff [s]_\sim \models_{\mathcal{D}/\sim} \Phi$ *for all PCTL* formulae* Φ.

This implies $\mathcal{D} \models \Phi$ if and only if $\mathcal{D}/\!\sim \models \Phi$ for all PCTL* formulae Φ. Put differently, a given formula may be verified on the (possibly much smaller) bisimulation quotient while preserving the verification result of the original model.

The Prism modelling language. Prism is a widely used, state-of-the-art probabilistic model checker. It features a high-level modelling language, adopted by several other tools. We focus on a subset of this language whose semantics corresponds to DTMCs. Let *Var* be a finite set of variables, each of which is typed as either Boolean or bounded integer, and $\Sigma(Var)$ be the set of all valuations of variables in *Var* that respect these types. Furthermore, let $Expr_{Var}$ and $BExpr_{Var}$ denote the set of all expressions over *Var* and the Boolean expressions thereof, respectively. For $e \in Expr_{Var}$ and $s \in \Sigma(Var)$, let $[\![e]\!]_s$ be the value of e in s, i.e. the value of e when all occurring variables are replaced by their values in s. Let $s \models b$ for $b \in BExpr_{Var}$ iff $[\![b]\!]_s = 1$ and let $[\![b]\!]$ be the set of all valuations that assign the truth value 1 to b, i.e., $[\![b]\!] = \{s \in \Sigma(Var) \mid s \models b\}$.

Definition 5 (Assignment). *An* assignment *over a set Var of variables is a function* $E : Var \to Expr_{Var}$ *that complies with the respective types.*

For a valuation $s \in \Sigma(Var)$ and an assignment E over *Var*, we write $s \xrightarrow{E} s'$ if and only if for all $v \in Var$ we have $[\![v]\!]_{s'} = [\![E(v)]\!]_s$ with the intuition that s is transformed into s' by updating the values of all variables according to E. If $s \xrightarrow{E} s'$, we will say that the execution of E in s leads to state s'.

Definition 6 (Guarded command). *A* guarded command $c = (a, g, (p_1, E_1), \ldots, (p_n, E_n))$ *over Var and a set Act of actions consists of (i) an action* $a \in Act$, *(ii) a guard* $g = guard(c) \in BExpr_{Var}$ *and (iii) probabilities* $p_i \in [0, 1]$ *associated with assignments* E_i *over Var for* $1 \leq i \leq n$ *such that* $\sum_{1 \leq i \leq n} p_i = 1$.

Syntactically, we write c as $[a]\ g \longrightarrow p_1 : (Var'{=}E_1) + \ldots + p_n : (Var'{=}E_n)$, where $Var'{=}E_i$ is short for a list of entities of the form $v'{=}e$ for $v \in Var$ and $e = E_i(v)$. Intuitively, a guarded command can be executed in every state that satisfies its guard. If it is executed, the ith assignment is executed with probability p_i. A probabilistic program comprises a set of guarded commands.

$pc\colon int[1,4]$ init 1; $\quad h, f, r\colon bool$ init $false$;

$[coin]\ pc{=}1 \ \rightarrow 0.5 : (pc'{=}pc+1)\&(h'{=}\neg h) + 0.5 : (pc'{=}pc+1)\&(h'{=}h)$;

$[proc]\ pc{=}2 \ \rightarrow 0.2 : (pc'{=}pc+1)\&(f'{=}\neg f) + 0.8 : (pc'{=}pc+1)\&(f'{=}f)$;

$[rtn1]\ pc{=}3 \wedge h \wedge \neg f \ \rightarrow 0.2 : (pc'{=}pc+1)\&(r'{=}0)\&(f'{=}1) + 0.8 : (pc'{=}pc+1)\&(r'{=}1)$;

$[rtn2]\ pc{=}3 \wedge \neg h \wedge \neg f \ \rightarrow 0.5 : (pc'{=}pc+1)\&(r'{=}0)\&(f'{=}1) + 0.5 : (pc'{=}pc+1)\&(r'{=}1)$;

$[rest]\ pc{=}3 \wedge f \ \rightarrow 0.99 : (pc'{=}1)\&(h'{=}0)\&(f'{=}0)\&(r'{=}0) + 0.01 : (pc'{=}pc+1)$;

$[done]\ pc{=}4 \ \rightarrow 1 : (pc'{=}pc)$;

Fig. 1. Running example: The probabilistic program P_{Ex}

Definition 7 (Probabilistic program). *A* probabilistic program $P = (Var, s_{init}, Act, Comm)$ *consists of (i) a finite set* Var *of variables, (ii) an initial state* $s_{init} \in \Sigma(Var)$*, (iii) a finite set* Act *of actions and (iv) a finite set* $Comm$ *of guarded commands over* Var *and* Act*. (v) Additionally, for each* $s \in \Sigma(Var)$*, there must exist exactly one* $c \in Comm$ *with* $s \models guard(c)$*.*

Example 1. Fig. 1 shows our running example: a probabilistic program P_{Ex} over variables $Var_{Ex} = \{pc, h, f, r\}$, modelling a randomised algorithm with 4 phases (indicated by pc). The algorithm starts by throwing a coin (h) before a processing step that has certain probability (0.2) to fail (f). In case of a failure, the algorithm restarts with high probability (0.99) and terminates in error otherwise. If there is no failure, it returns a result r that is either true or false with a certain probability that depends on the coin flip. As false is the (supposedly) incorrect result, the fail flag is set in this case. Note that all variables v that are not assigned any value in an assignment keep their previous value, i.e. have $v' = v$. For the sake of clarity, we sometimes include these superfluous assignments. In further examples in this paper, we refer to the command with name a by c_a (for instance, the coin flip command will be referred to as c_{coin}) and we denote the ith assignment of command c_a by $E_{a,i}$.

The PRISM modelling language also supports parallel composition of modules, where some commands are executed synchronously. We deal with such models by flattening them into one module, using a symbolic composition of the commands that need to synchronise. While this may increase the number of commands in the program, this is always possible in a totally automatic way and is thus no restriction on the expressivity of the probabilistic programs considered.

The semantics of a probabilistic program P is a DTMC $[\![P]\!]$ whose state space is $\Sigma(Var)$ and whose transitions are defined by the guarded commands. The additional constraint (see Def. 7 (v)) assures that a guarded command induces a probability distribution over the successor states and that there are neither deadlock states nor states that have multiple guarded commands enabled. Note that this is no restriction. If there exists a state that satisfies no guard, the state has no outgoing transition and is thus equipped with a self-loop for model checking purposes. This can already be done at the language level by introducing a loop command for states that do not have any outgoing transition. On the other hand, if there is a state that satisfies multiple guards, this corresponds to a nondeterministic choice in that state. Hence, the semantics of the program would be

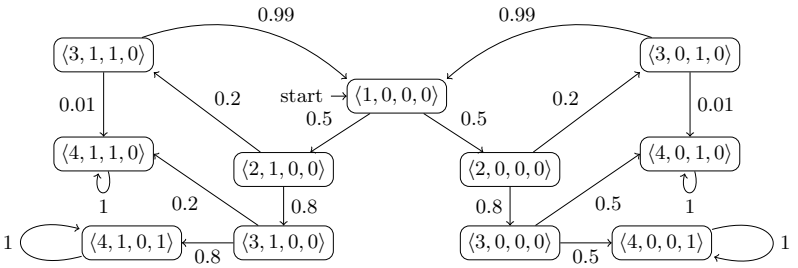


Fig. 2. The (reachable) fragment of $[\![P_{Ex}]\!]$, with states of the form $\langle pc, h, f, r\rangle$

an MDP instead of a DTMC. The extension of our approach to non-deterministic models is possible (see Section 4), but not the main concern of this paper.

Example 2. The reachable part of the state space of the probabilistic program P_{Ex} (see Fig. 1) is depicted in Fig. 2, where the states are of the form $\langle pc, h, f, r\rangle$.

3 SMT-Based Bisimulation Minimisation

We now give our SMT-based approach to bisimulation minimisation. We first summarise how to perform minimisation using partition refinement, and then describe its symbolic implementation using weakest preconditions and SMT.

Partition Refinement. The standard approach to deriving a bisimulation quotient algorithmically prior to verification is to use partition refinement [5,18]. This technique starts with an initially coarse partition of the state space S and successively splits (refines) blocks containing states that have different behaviour with respect to the current partition until no more refinement is necessary. If the initial partition is chosen based on the atomic propositions, this results in the coarsest partition of S that induces a bisimulation, i.e., $S/\!\sim$.

In the probabilistic setting, a block B of partition Π needs to be split if it contains two states that possess different quotient distributions with respect to the current partition, i.e. if there exist $s_1, s_2 \in B$ such that $[\mathbf{P}(s_1, \cdot)]_{R(\Pi)} \neq [\mathbf{P}(s_2, \cdot)]_{R(\Pi)}$ where $R(\Pi)$ is the equivalence relation induced by the partition Π. In other words, in order to implement a partition refinement approach, we need to: (i) determine if B contains states with different quotient distributions and (ii) if so, identify the subsets of B which agree on the quotient distribution, because these form the blocks into which B is split.

In our work, a partition $\Pi = \{B_1, \ldots, B_k\}$ of the state space $S = \Sigma(Var)$ is represented *symbolically* by corresponding Boolean expressions $\pi = \{b_1, \ldots, b_k\}$ such that $B_i = [\![b_i]\!]$ for each $1 \leq i \leq k$. In the sections below, we describe how to reason symbolically, using weakest preconditions, about: (i) whether a state's successors are contained in a particular block; and (ii) the quotient distribution of a state. Once the partition refinement algorithm terminates (i.e., there are no further blocks in the current partition to split), the quotient DTMC is constructed as follows: its state space is taken as the set Π of blocks in the final

partition; and the transition probabilities for each block $B \in \Pi$ are then given by the (unique) corresponding quotient distribution for that block.

Weakest Preconditions. To reason symbolically about the effect of an assignment E in a command of a probabilistic program, we use the weakest precondition operation. The weakest precondition of $b_i \in BExpr_{Var}$ with respect to E, denoted $wp(b_i, E)$, characterizes exactly the valuations s of $\Sigma(Var)$ for which the successor valuation after assignment E satisfies b_i:

$$s \models wp(b_i, E) \quad \iff \quad s \xrightarrow{E} s' \text{ with } s' \models b_i.$$

We can determine $wp(b_i, E)$ through a purely syntactic modification of b_i by simultaneously replacing each occurrence of each variable $v \in Var$ in b_i by $E(v)$.

Example 3. Consider the first assignment of the command c_{coin} in Example 1:

$$E_{coin,1}(pc) = pc + 1,\ E_{coin,1}(h) = \neg h,\ E_{coin,1}(f) = f,\ E_{coin,1}(r) = r$$

and let $b_1{=}\neg h$. Then $wp(b_1, E_{coin,1}) = \neg\neg h \equiv h$. This reflects the fact that exactly the states s with $s \models h$ are transformed into a state s' by $E_{coin,1}$ such that $s' \models \neg h$. Intuitively, this is because $E_{coin,1}$ flips the truth value of h. □

We fix, from now on, a command $c = [a]\ g \longrightarrow p_1 : (Var'{=}E_1) + \ldots + p_n : (Var'{=}E_n)$ with n assignments. Given n Boolean expressions $b_{i_1}, \ldots, b_{i_n}$ for indices $i_1, \ldots, i_n \in \{1, \ldots, k\}$, observe that, for $s \in \Sigma(Var)$:

$$s \models \bigwedge\nolimits_{j=1}^{n} wp(b_{i_j}, E_j) \quad \iff \quad \text{for all } 1 \le j \le n \ .\ s \xrightarrow{E_j} s_j \text{ such that } s_j \models b_{i_j}\ .$$

Note, however, that the command c is not necessarily enabled in all the states s since some might fail to satisfy the guard g of c. If, in addition, such a state satisfies g, we know that in the semantics of the probabilistic program, state s has an outgoing probability distribution that goes with probability p_j to a state s_j satisfying b_{i_j}. We say that the jth assignment of c will lead into block B_{i_j}, which we write as $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$. If, on the other hand, $\varphi = g \wedge \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$ is unsatisfiable, i.e. there is no $s \in \Sigma(Var)$ such that $s \models \varphi$, then there exists no state s for which $s \xrightarrow{E_j} s_j$ such that $s_j \models b_{i_j}$ for all $1 \le j \le n$.

Example 4. Consider the command c_{coin} and its two assignments $E_{coin,1}$ and $E_{coin,2}$ (see Fig. 1) and let $b_1 = \neg h$ and $b_2 = h$. Then:

$$\bigwedge\nolimits_{i=1}^{2} wp(b_i, E_{coin,i}) = \neg\neg h \wedge h \equiv h\ .$$

So, for a state s where $s \models guard(c_{coin})$, i.e. $s \models (pc = 1)$, and $s \models h$, we conclude that command c is enabled, assignment $E_{coin,1}$ will lead into a state satisfying $\neg h$ (b_1) and assignment $E_{coin,2}$ will lead into one satisfying h (b_2). We denote this by $s \xrightarrow{c_{coin}} (B_1, B_2)$. □

Quotient Distributions. Reasoning in a similar way, we can also determine the quotient distribution for a state s with respect to the current partition. From above, if $s \in \Sigma(Var)$ with $s \models guard(c)$ and $s \models \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$, we have that $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$. Since command c has n assignments, each leading into a block, we have an n-tuple $(B_{i_1}, \ldots, B_{i_n})$ of target blocks per state. These blocks B_{i_j} are not necessarily distinct. Accordingly, to determine the quotient probability distribution of s with respect to the partition Π, we sum up the probabilities that lead into the same blocks.

Note, however, that the probabilities do not appear in the formulas. Depending on whether or not a state satisfies the conjunction of weakest preconditions for certain blocks, we know whether the corresponding assignments will take that state to the associated blocks. Based on this knowledge, the probability distribution is directly given by the probabilistic program.

Example 5. Consider again Example 4 and note that $b_1 = \neg h$ and $b_2 = h$ induce a partition of $\Sigma(Var)$. From the (fixed) probabilities associated with the two assignments, $E_{coin,1}$ and $E_{coin,2}$ in the program (Fig. 1), we can conclude that, for all states s with $s \models (pc = 1)$ and $s \models h$, there is a 0.5 probability to move to block B_1 in the next step and the same holds for B_2.

In contrast, consider the partition Π' induced by $b'_1 = (pc \neq 2)$ and $b'_2 = (pc = 2)$. Then all states satisfying $guard(c_{coin}) = (pc = 1)$ and

$$\bigwedge\nolimits_{i=1}^{2} wp(b'_2, E_{coin,i}) = (pc + 1 = 2) \wedge (pc + 1 = 2) \equiv (pc = 1)$$

will move to B'_2 with both assignments. In other words, the quotient probability distribution for all states s with $s \models (pc = 1)$ is given by $\mathbf{P}(s, B'_2) = 1.0$ and $\mathbf{P}(s, B'_1) = 0.0$. This can also be seen by looking at the probabilistic program: starting in a state with $(pc = 1)$ will result in a state with $(pc = 2)$ with probability 1.0 after one step, because only c_{coin} is available in these states and all assignments of that command increase pc by one. □

SMT-Based Block Refinement. Recall from the start of this section that the key operation required to perform bisimulation minimisation using partition refinement is to split a block B by identifying the different possible quotient distributions for states within B. We now explain, building on the above, how to perform this using queries executed by an SMT solver.

Suppose again, that the current partition is $\Pi = \{B_1, \ldots, B_k\}$, where each block B_i is represented by a Boolean expression b_i, i.e. $B_i = [\![b_i]\!]$, and that we are to refine block $B = [\![b]\!] \in \Pi$. We now formulate this computation step as a series of queries to an SMT solver. Given a command $c = [a]\ g \longrightarrow p_1 : (Var'{=}E_1) + \ldots + p_n : (Var'{=}E_n)$ as before, observe that there is a state $s \in B$ where c is enabled and the jth assignment of c leads into block B_{i_j} for all $1 \leq j \leq n$, i.e. $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$, if and only if the formula

$$b \wedge g \wedge \bigwedge\nolimits_{j=1}^{n} wp(b_{i_j}, E_j) \tag{1}$$

is satisfiable. While the first conjunct ensures that the state is in block B, the rest of the formula guarantees that the state in question exhibits the appropriate behaviour.

Example 6. Recall the partition given by $b_1{=}\neg h, b_2{=}h$ of Example 4. Since:

$$\underbrace{\neg h}_{b_1} \wedge \underbrace{(pc=1)}_{guard(c_{coin})} \wedge \underbrace{\neg h}_{wp(b_2,E_{coin,1})} \wedge \underbrace{h}_{wp(b_2,E_{coin,2})}$$

is unsatisfiable, we can conclude that there are no states in B_1 that have c_{coin} enabled such that both assignments lead into block B_2. Intuitively, this is because the first assignment flips the value of h while the second one leaves the value untouched. Therefore, there exists no possible value of h such that after executing either one of the assignments h (b_2) always holds. In contrast, because:

$$\underbrace{\neg h}_{b_1} \wedge \underbrace{(pc=1)}_{guard(c_{coin})} \wedge \underbrace{\neg h}_{wp(b_2,E_{coin,1})} \wedge \underbrace{\neg h}_{wp(b_1,E_{coin,2})} \equiv \neg h \wedge (pc=1)$$

is satisfiable, we know that there is a state $s \in B_1$ with $s \xrightarrow{c_{coin}} (B_2, B_1)$. Furthermore, these states are exactly characterized by $\neg h \wedge (pc = 1)$. □

Now, we can determine all possible target block tuples (and hence quotient distributions) for a block B under c by checking the corresponding formulas for satisfiability. The problem with this naive approach is the sheer number of satisfiability queries. Consider a guarded command with n assignments and a partition of k blocks. Then there are n^k different block tuples the command might lead into, and we would therefore need as many SMT queries in order to determine all target block tuples, despite the fact that almost all of them will be unsatisfiable. This observation justifies the key idea of our approach: we determine the different quotient probability distributions available in B via c using an ALLSAT enumeration query answered by an SMT solver. That is, we present a formula system to the SMT solver whose solutions correspond to available target block tuples and let the SMT solver enumerate all possible solutions. This means that we only need to create the formula system once and the solver will only enumerate as many solutions as there are.

To that end, we construct a formula system $S_{\Pi,c}$ that uses unique auxiliary variables $z_{i,j}$ for each weakest precondition $wp(b_i, E_j)$ with the intention that the values of these variables in a satisfying assignment encode the behaviour of some states in B. More concretely, we assert that $z_{i,j}$ implies $wp(b_i, E_j)$ and, additionally, for each assignment E_j we require at least one of the corresponding $z_{i,j}$ to be true. This results in the following formula system $S_{\Pi,c}$:

$$b \tag{2}$$

$$g \tag{3}$$

$$z_{i,j} \rightarrow wp(b_i, E_j) \text{ for all } 1 \leq i \leq k \text{ and all } 1 \leq j \leq n \tag{4}$$

$$\bigvee_{i=1}^{k} z_{i,j} \text{ for all } 1 \leq j \leq n \tag{5}$$

Observe that this yields a correspondence between satisfying assignments for the variables $z_{i,j}$ and target block tuples under command c available in B as follows.

In general, given a satisfying assignment α of all variables in this system, (5) guarantees that there exist indices $1 \leq i_1, \ldots, i_n \leq k$ such that $\alpha(z_{i_1,1}) = \ldots = \alpha(z_{i_n,n}) = 1$. Then, because of (4), it follows that $s \models \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$. Together with (2) and (3) this implies $s \models b \wedge g \wedge \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$ (cf. formula (1)). In other words, we can conclude that there exists $s \in B$ with $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$. Note that we are only interested in the values of the $z_{i,j}$ in a satisfying assignment returned by the solver, as they alone enumerate the target block tuples. Also, in (4), implications rather than equivalences suffice, because $wp(b_{i_k}, E_j)$ and $wp(b_{i_l}, E_j)$ are mutually unsatisfiable for $i_k \neq i_l$. Intuitively, this is because the jth assignment cannot lead to two different blocks B_{i_k} and B_{i_l} from one state. Hence, because of (5), in each satisfying assignment exactly one of the $z_{i,j}$ is assigned true for all $1 \leq j \leq n$ and each solution corresponds to one target block tuple. The solution found can easily be ruled out by additionally asserting $\bigwedge_{j=1}^{n} \neg z_{i_j,j}$ and the solver can then be used to retrieve the next solution if there is any.

Example 7. Assume the solver returns a solution (i.e. a valuation $s \in \Sigma(Var)$ such that all formulas evaluate to true) to the formula system $S_{\Pi,c}$ with $z_{1,j} = 1$ for all $1 \leq j \leq n$. Obviously $s \in B$, because $s \models b$. Then, because of (3), the command c is enabled in s. Also, due to (4), we have $s \models \bigwedge_{j=1}^{n} wp(b_1, E_j)$. Stated differently, there exists an $s \in B$ such that $s \xrightarrow{c} (B_1, \ldots, B_1)$.

The SmtRefine **algorithm.** Algorithm 1 presents an abstract implementation of SmtRefine, our Smt-based block refinement procedure. It takes as input a partition Π of $\Sigma(Var)$ given by Boolean expressions and a block $B \in \Pi$ given by Boolean expression b. It returns a partition Π_B of B given by Boolean expressions, such that all states in each block of Π_B share the same quotient distribution wrt. Π. In other words, Π_B is a stable partition of B wrt. Π.

The algorithm computes a (partial) mapping $sig\colon Dist_\Pi \to 2^{BExpr_{Var}}$ from $Dist_\Pi$, the probability distributions over Π, to a set of (mutually unsatisfiable) Boolean expressions. This is done such that a state $s \in B$ has the quotient probability distribution μ iff it satisfies one of the expressions in $sig(\mu)$, i.e.:

$$[\mathbf{P}(s,\cdot)]_{R(\Pi)} = \mu \quad \Longleftrightarrow \quad \text{there exists } b_\mu \in sig(\mu) \text{ such that } s \models b_\mu \tag{6}$$

where $R(\Pi)$ is the equivalence relation induced by the partition Π. Put differently, sig maps all available quotient probability distributions in B to Boolean expressions that characterize exactly the states having these distributions. Then, upon termination of SmtRefine, Π_B is given by the expressions:

$$\left\{ \bigvee_{b_\mu \in sig(\mu)} b_\mu \;\mid\; \mu \in keys(sig) \right\} \tag{7}$$

where $keys(sig)$ denotes the domain of sig, i.e. the quotient distributions in B.

The algorithm works as follows. We start by initialising the mapping sig to the empty mapping. Then, for each command c in the probabilistic program, we build the formula system $S_{\Pi,c}$ and pass it to the solver ($assert(S_{\Pi,c})$). As long as the solver finds any solutions (hasNextSolution()), we retrieve that solution, say α, via $getSolution()$ from the SMT solver. Note that α is a mapping of all the variables in Var and the auxiliary variables $z_{i,j}$ to their respective domains such that all formulas of $S_{\Pi,c}$ evaluate to true. As we are only interested in the values of the variables $z_{i,j}$, we can drop the other parts of the assignment.

As previously shown, such a solution implies that there exists a state $s \in B$ with $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$ for certain indices i_j, $1 \leq j \leq n$. Given a solution α to the formula system, the function $getBlockCombination(\alpha)$ can compute these indices. As there may be different assignments of c leading s into the same block, we need to determine the quotient distribution μ by summing up the corresponding probabilities given by c. This is done by the function $compDist()$, which obviously needs to take the target block tuple as a parameter. Now that we know that there exists (at least) one state $s \in B$ whose quotient distribution is μ, we need to update the mapping sig accordingly. This is done by adding to $sig(\mu)$ the expression characterizing exactly those states that lead into the current target block tuple via c. Finally, we rule out the solution α previously found by the solver. More precisely, we rule out that particular combination of the $z_{i,j}$ being set to 1, because we do not want to enumerate this target block tuple again. If the solver now still finds possible target block tuples, we repeat the whole procedure. Note that the while-loop in Alg. 1 realises an ALLSAT procedure, as all solutions of the formula system are first found (via $getSolution()$) and ruled out later (via $ruleOutSolution(\alpha)$) as long as there exists another solution. After all target block tuples have been enumerated, we need to determine whether the block needs to be split. This is the case, if there is more than one quotient probability distribution available in B, i.e. if the domain of sig consists of more than one element. In that case, a stable partition of B is given by sig according to equation (7). If there is exactly one quotient distribution available, we don't need to split B and just return b itself.

The correctness of the SMT-based refinement algorithm is given by:

Theorem 1 (Correctness). *Given a partition $\Pi = \{B_1, \ldots, B_k\}$ represented by a set of Boolean expressions $\{b_1, \ldots, b_k\}$ and a block $B = [\![b]\!] \in \Pi$,* SMTREFINE *returns a partition Π_B of B given by mutually unsatisfiable Boolean expressions $\{b'_1, \ldots, b'_m\}$ such that for all $s_1, s_2 \in B$:*

$$\mathbf{P}(s_1, T) = \mathbf{P}(s_2, T) \text{ for all } T \in S/\Pi \quad \textit{iff} \quad s_1 \models b'_i \Leftrightarrow s_2 \models b'_i \text{ for all } 1 \leq i \leq m$$

Example 8. Let P_{Ex} be the probabilistic program in Fig. 1. Furthermore, let the initial partition be $\Pi_{init} = \{[\![pc \neq 4]\!], [\![pc = 4]\!]\}$ given by the Boolean expressions $b_1 = (pc \neq 4)$ and $b_2 = (pc = 4)$. Now assume that the first block that is to be refined is $B_1 = [\![b_1]\!]$.

For the outermost loop in Alg. 1, we consider the commands in the order in which they appear in P_{Ex}. Accordingly, we start with c_{coin} and build the formula system $S_{\Pi,c_{coin}}$ as:

Algorithm 1. SMT-based block refinement

Require: partition Π given by $\pi = \{b_1, \ldots, b_k\}$, block $B = [\![b]\!] \in \Pi$
Ensure: returns stable (wrt. Π) partition Π_B of B

procedure SMTREFINE($b, \pi = \{b_1, \ldots, b_k\}$) ▷ Refines B wrt. Π
  $sig = \emptyset$ ▷ Initialize mapping of $Dist_\Pi$ to set of expressions in $BExpr_{Var}$
  for each $c = [a]\ g \longrightarrow p_1 : Var' = E_1 + \ldots + p_n : Var' = E_n \in Comm$ **do**
    $assert(S_{\Pi,c})$
    while hasNextSolution() **do**
      $\alpha \leftarrow getSolution()$ ▷ Retrieve solution from solver
      $(B_{i_1}, \ldots, B_{i_n}) \leftarrow getBlockCombination(\alpha)$ ▷ Compute a target block combination induced by α
      $\mu \leftarrow compDist(B_{i_1}, \ldots, B_{i_n})$ ▷ Compute the corresponding distribution
      $sig(\mu) \leftarrow sig(\mu) \cup \{b \wedge g \wedge \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)\}$ ▷ Update signature mapping
      $ruleOutSolution(\alpha)$ ▷ Rule out current solution for solver
    end while
  end for

  if $|keys(sig)| > 1$ **then**
    return $\{ \bigvee_{b_\mu \in sig(\mu)} b_\mu \mid \mu \in keys(sig) \}$ ▷ Split block only if necessary
  else
    return $\{b\}$ ▷ Otherwise return input block
  end if
end procedure

$$\underbrace{pc \neq 4}_{b_1} \qquad \underbrace{pc = 1}_{guard(c_{coin})}$$

$$z_{1,1} \rightarrow \underbrace{pc + 1 \neq 4}_{wp(b_1, E_{coin,1})} \qquad z_{1,2} \rightarrow \underbrace{pc + 1 = 4}_{wp(b_2, E_{coin,1})}$$

$$z_{2,1} \rightarrow \underbrace{pc + 1 \neq 4}_{wp(b_1, E_{coin,2})} \qquad z_{2,2} \rightarrow \underbrace{pc + 1 = 4}_{wp(b_2, E_{coin,2})}$$

$$z_{1,1} \vee z_{1,2}$$

$$z_{2,1} \vee z_{2,2}$$

As the formula $pc = 1$ is part of the formula system, the value of pc is fixed to one, which in turn means that $z_{1,2}$ and $z_{2,2}$ can never be set to true in a solution of the system. This corresponds to the fact that all states that have c_{coin} enabled (and therefore need to satisfy its guard, namely $pc = 1$) have no way of going to block B_2 with any of the assignments, because pc is only increased by 1.

In fact, the solver only returns a solution with $z_{1,1} = 1$ and $z_{2,1} = 1$, meaning that, for all states in B_1 that have c_{coin} enabled, both assignments lead into B_1. Accordingly, the quotient distribution μ_1 is given by:

$$\mu_1(B_1) = 1.0 \text{ and } \mu_1(B_2) = 0.0$$

We update the previously empty mapping sig to:

$$sig = \{\mu_1 \mapsto \{ \underbrace{pc \neq 4 \wedge pc = 1}_{s \in B_1 \wedge s \models guard(c_{coin})} \wedge \underbrace{pc+1 \neq 4 \wedge pc+1 \neq 4}_{s \models wp(b_1, E_{coin,1}) \wedge wp(b_1, E_{coin,2})} \}\}$$

and continue with constructing the formula system for command c_{proc}:

$$\begin{aligned}
&pc \neq 4\\
&pc = 2\\
&z_{1,1} \rightarrow pc+1 \neq 4 \qquad z_{1,2} \rightarrow pc+1 = 4\\
&z_{2,1} \rightarrow pc+1 \neq 4 \qquad z_{2,2} \rightarrow pc+1 = 4\\
&z_{1,1} \vee z_{1,2}\\
&z_{2,1} \vee z_{2,2}
\end{aligned}$$

Except for the guard it is exactly as the previous formula system, because the assignments of this command do exactly the same transformation to pc as the assignments of c_{coin}. Exactly because of this, the solver will once again only return a solution with $z_{1,1} = 1$ and $z_{2,1} = 1$, which means that the corresponding states also possess the same quotient distribution μ_1 as before. This yields:

$$\begin{aligned}
sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc+1 \neq 4 \wedge pc+1 \neq 4,\\
& \qquad pc \neq 4 \wedge pc = 2 \wedge pc+1 \neq 4 \wedge pc+1 \neq 4\} \qquad \}
\end{aligned}$$

The next command to consider is c_{rtn1}. The formula system looks as follows:

$$\begin{aligned}
&pc \neq 4\\
&pc = 3 \wedge h \wedge \neg f\\
&z_{1,1} \rightarrow pc+1 \neq 4 \qquad z_{1,2} \rightarrow pc+1 = 4\\
&z_{2,1} \rightarrow pc+1 \neq 4 \qquad z_{2,2} \rightarrow pc+1 = 4\\
&z_{1,1} \vee z_{1,2}\\
&z_{2,1} \vee z_{2,2}
\end{aligned}$$

This time, the solver will return $z_{1,2} = z_{2,2} = 1$ as the only solution. Intuitively, this is due to the fact that all states satisfying the guard of c_{rtn1} must have $pc = 3$, which means that, after executing either one of the assignments, the value of pc will be 4 and thus will lead to a state in block B_2. As both updates lead to B_2, the corresponding quotient distribution in these states is given by:

$$\mu_2(B_1) = 0.0 \text{ and } \mu_2(B_2) = 1.0$$

which results in:

$$\begin{aligned}
sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc+1 \neq 4 \wedge pc+1 \neq 4,\\
& \qquad pc \neq 4 \wedge pc = 2 \wedge pc+1 \neq 4 \wedge pc+1 \neq 4\},\\
& \mu_2 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge h \wedge \neg f\} \qquad \}\,.
\end{aligned}$$

Apart from the guard, the formula system for the next command c_{rtn2} is the same as the one before and also has the same solution, which means that these states also have μ_2 as the outgoing quotient distribution. This updates sig to:

$$\begin{aligned} sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4, \\ & \qquad pc \neq 4 \wedge pc = 2 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4\}, \\ & \mu_2 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge h \wedge \neg f, \\ & \qquad pc \neq 4 \wedge pc = 3 \wedge \neg h \wedge \neg f\} \qquad \} . \end{aligned}$$

The next command is c_{rest}, which leads to $S_{\Pi,c_{rest}}$ as follows:

$$\begin{aligned} & pc \neq 4 \\ & pc = 3 \wedge f \\ & z_{1,1} \rightarrow 1 \neq 4 \qquad\qquad z_{1,2} \rightarrow 1 = 4 \\ & z_{2,1} \rightarrow pc + 1 \neq 4 \qquad z_{2,2} \rightarrow pc + 1 = 4 \\ & z_{1,1} \vee z_{1,2} \\ & z_{2,1} \vee z_{2,2} \end{aligned}$$

for which the solver identifies $z_{1,1} = 1$ and $z_{2,2} = 1$ as the only solution. Intuitively, this says that for all states $s \in B_1$ that have this command enabled, the first assignment will lead back into B_1 while the second assignment leads into B_2, which is obvious considering that the first assignment resets pc to 1 and the second assignment increases pc from 3 to 4. This means that all states $s \in B_1$ with $s \models guard(c_{rest})$ possess the quotient distribution μ_3 with:

$$\mu_3(B_1) = 0.99 \text{ and } \mu_3(B_2) = 0.01$$

which updates sig to:

$$\begin{aligned} sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4, \\ & \qquad pc \neq 4 \wedge pc = 2 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4\}, \\ & \mu_2 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge h \wedge \neg f, \\ & \qquad pc \neq 4 \wedge pc = 3 \wedge \neg h \wedge \neg f\}, \\ & \mu_3 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge f \wedge 1 \neq 4 \wedge pc + 1 = 4\} \qquad \} . \end{aligned}$$

Finally, for the last command, namely c_{done}, the formula system is as follows:

$$\begin{aligned} & pc \neq 4 \qquad\qquad pc = 4 \\ & z_{1,1} \rightarrow pc \neq 4 \qquad z_{1,2} \rightarrow pc = 4 \\ & z_{2,1} \rightarrow pc \neq 4 \qquad z_{2,2} \rightarrow pc = 4 \\ & z_{1,1} \vee z_{1,2} \\ & z_{2,1} \vee z_{2,2} \end{aligned}$$

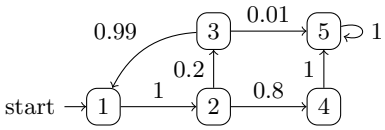


Fig. 3. The bisimulation quotient $[\![P_{Ex}]\!]/\sim$

where already the two formulas in the first line are (together) unsatisfiable. This reflects the fact that no state in $B_1 = [\![pc \neq 4]\!]$ has this command enabled, because its guard requires pc to be equal to 4. Therefore sig is not updated and as there are no more commands to consider, the loop terminates.

Consequently, B needs to be split into three sub-blocks according to the different quotient distributions recorded in sig. Continuing this for all blocks, we compute the stable partition (expressions have been simplified to improve readability):

$$\pi_{Ex}^{sta} = \{\underbrace{pc = 1 \wedge \neg f}_{B_1^{sta}}, \underbrace{pc = 2 \wedge \neg f}_{B_2^{sta}}, \underbrace{pc = 3 \wedge f}_{B_3^{sta}},$$
$$\underbrace{pc = 3 \wedge \neg f}_{B_4^{sta}}, \underbrace{pc = 4}_{B_5^{sta}}, \underbrace{pc = 1 \wedge f}_{B_6^{sta}}, \underbrace{pc = 2 \wedge f}_{B_7^{sta}}\}.$$

For this final partition, we extract the quotient DTMC depicted in Fig. 3. Note that the distributions for the blocks can easily be kept track of during the refinement steps and are thus known. Also note that we already omitted the unreachable blocks B_6^{sta} and B_7^{sta}. □

4 Experiments

Implementation. We implemented a C++-based prototype of our algorithm, using Microsoft's Z3 as the backend SMT solver and comprising about 5000 lines of code. We restrict our attention to PRISM models in which expressions involve linear integer arithmetic, which is typically the case in practice. Also, we use some optimisation techniques to the approach previously described, one of which passes an additional conjunct to the solver that rules out some unreachable blocks in the refinement procedure.

Case Studies. We evaluated our implementation on a set of probabilistic model checking benchmarks,[1] running our experiments on a Core i7 processor with 2.6GHz, and limited to 5GB RAM and 48 hours of runtime. If the experiment ran out of time or memory, this is marked as TO and MO, respectively. For the comparisons we considered all three engines of PRISM (hybrid, sparse and MTBDD) and give the times for the default (hybrid) and the respective best

[1] All models are available from `http://www.prismmodelchecker.org/casestudies/`.

Table 1. Results for the synchronous leader election protocol

		original DTMC					quotient DTMC		factor	
N	K		hybrid (default)		best				low	high
		states	constr.	verif.	constr.	verif.	states	constr.		
4	9	19817	6.00	88.75	5.99	3.18	10	11.40	0.81	7.78
4	11	44107	41.74	543.85	23.91	14.20	10	26.57	1.43	22.04
5	9	236745	414.91	9456.78	483.86	13.48	12	497.91	1.0	18.99
5	11	644983	4083.82	60784.22	3695.16	45.60	12	1945.99	1.92	33.33
6	9	2659233	TO	TO	53695.52	643.97	14	28548.85	1.90	–

engine for the corresponding example for both construction (column *constr.*) as well as verification (column *verif.*). Please note that the quotient DTMCs were first computed by our prototype as well as verified by PRISM afterwards, but as the verification of the quotient took negligible time (i.e. less than 5ms) for all our experiments, we omitted these entries in the tables.

As it is already known that bisimulation minimisation can lead to drastically smaller state spaces, the key point we want to compare is not the model size in terms of states, but the time needed to verify the properties of interest. For this reason, we list time reduction factors: they are the ratio of the total time consumption of PRISM for the given model to the total time needed to minimise the model using our prototype and verify it afterwards using PRISM, where the verification, as explained earlier, took virtually no time at all. Since the time PRISM needs for the full model strongly depends on the engine used, we list two reduction factors, where the lower is computed with respect to the best and the higher with respect to the worst PRISM engine for the given model.

Synchronous leader election protocol. In Itai and Rodeh's protocol, N processors each probabilistically chose one of K different values to pass synchronously around a ring in order to determine a unique leader. We computed the probabilities that a leader is eventually elected and that a leader is elected within L rounds, for $L = 3$. The results are shown in Table 1. The construction and verification of the bisimulation quotient is about as fast as the best (i.e., sparse) PRISM engine. However, in comparison to the default PRISM engine, we achieve substantial speed-ups using SMT-based quotienting. Note that the quotient system can be used to verify the step-bounded property for arbitrary values of L, as it preserves all PCTL* formulae and does not depend on L. So, verifying this property for more values of L will increase the reduction factors roughly linearly.

Crowds protocol. Reiter and Rubin's Crowds protocol aims to send a message anonymously to a destination by randomly routing it R times through a crowd of size N. For this model, we restricted the reachable blocks by over-approximating the reachable state space by an expression capturing the obvious fact that the total number of member observations cannot exceed the number of instances the

Table 2. Impact of bisimulation minimisation on the Crowds protocol model

		original DTMC					quotient DTMC		factor	
N	R		hybrid (default)		best				low	high
		states	constr.	verif.	constr.	verif.	states	constr.		
10	5	110562	0.17	0.48	0.17	0.29	73	1.13	0.58	1.76
10	20	$4.4 \cdot 10^9$	MO	MO	31.18	1623.54	313	21.11	78.39	–
20	5	2036647	174.97	111.59	180.80	8.14	73	2.67	70.71	107.25
20	20	?	MO	MO	MO	MO	313	56.26	–	–
25	5	5508402	MO	MO	MO	MO	73	3.74	–	–
25	20	?	MO	MO	MO	MO	313	80.03	–	–
500	5	?	MO	MO	MO	MO	73	8969.46	–	–
500	20	?	MO	MO	MO	MO	313	106724.27	–	–
600	5	?	MO	MO	MO	MO	73	18219.01	–	–
600	20	?	MO	MO	MO	MO	MO	MO	–	–

protocol has been run. Our model is a slight amendment of the model available on the PRISM website with less variables. Table 2 summarizes the results, where we computed the probability that the original sender was discovered more than once. Using our technique, we not only outperform PRISM in terms of runtime, but are also able to treat significantly larger model parameters. In fact, for the parameters where the state space size is indicated as unknown ("?"), using PRISM we were not able to build the state space let alone perform the actual verification.[2] Here, the crucial advantage of the SMT-based quotienting becomes apparent: since it avoids building the full state space of the original model, it shortens computation times while reducing the required memory.

Comparison with SIGREF. In addition to the comparison with PRISM, we compared our prototype to SIGREF, a tool that performs bisimulation minimisation symbolically on a BDD representation of the system [21]. We integrated SIGREF into PRISM in a way that works directly on the internal format of the model checker, which was possible because they share the MTBDD library CUDD. Table 3 illustrates the experimental results for both case studies, where the time reduction factor is the ratio of time needed for minimisation using SIGREF plus the verification using PRISM to the time needed for the minimisation using our SMT-based prototype plus the verification time using PRISM. Note that while the quotient DTMCs are isomorphic, the verification times differ between the two approaches, because the BDD representing the (same) system is different. For the first case study, we observe minor speed-ups compared to SIGREF. However, due to memory requirements, SIGREF was unable to minimise the state space of the last two models. For the Crowds protocol, note that SIGREF needs to build the full BDD representation of the state space prior to minimisation. As the time needed for model construction dominates the runtime, there is (almost) no

[2] We tried to build the state space of the smallest of these models (N=20, R=20) on a cluster with 196GB of memory, but aborted the experiment after one week.

Table 3. Comparison with SIGREF as the minimisation engine

(a) Synchronous leader election

N	K	constr.	verif.	red. factor
4	9	18.39	≈ 0	1.61
4	11	51.57	≈ 0	1.94
5	9	580.40	≈ 0	1.17
5	11	MO	MO	–
6	9	MO	MO	–

(b) Crowds protocol

N	R	constr.	verif.	red. factor
10	5	9.67	≈ 0	8.51
10	20	6100.093	90.26	293.26
20	5	481.022	≈ 0	180.02
20	5	MO	MO	–

scope for SIGREF to improve on PRISM's runtimes. Even worse, the additional intermediate BDDs prevented the minimisation for the parameters 20/5 under the memory restriction.

Possible Extensions. We conclude this section with an overview of several ways that our SMT-based approach to bisimulation can be extended.

Rewards. A Markov Reward Model (MRM) $(\mathcal{D}, r)$ is a DTMC $\mathcal{D}$ equipped with a function $r\colon S \to \mathbb{R}_{\geq 0}$ that assigns a non-negative real value to each state of $\mathcal{D}$. Upon passing through state s the reward $r(s)$ is gained, providing a quantitative measure in addition to the probabilities. In the PRISM modelling language, state-based rewards are defined in a similar fashion as commands, essentially attaching a reward expression e to all states satisfying a given Boolean expression b. If e is an expression evaluating to a constant, i.e., all states satisfying b share the same reward value, then rewards can be easily supported by our implementation by adjusting the initial partition appropriately.

Nondeterminism. In its current form, both the algorithm and the implementation treat only DTMCs and CTMCs and do not support their nondeterministic counterparts MDPs and CTMDPs, respectively. Our prototype can be extended in order to also support these models. For this, we lift the formula system $S_{\Pi,c}$ of section 3 to a system S_{Π} incorporating all commands with additional auxiliary variables x_c for $c \in Comm$ as follows:

$$b \tag{8}$$

$$x_c \leftrightarrow guard(c) \text{ for all } c \in Comm \tag{9}$$

$$z_{c,i,j} \to wp(b_j, E_{c,i}) \text{ for all } c \in Comm, \text{ all } 1 \leq j \leq k \text{ and all } 1 \leq i \leq |c| \tag{10}$$

$$x_c \to \bigvee\nolimits_{j=1}^{k} z_{c,i,j} \text{ for all } c \in Comm \text{ and } 1 \leq i \leq |c| \tag{11}$$

where the current partition $\Pi = [\![\pi]\!]$ is given by $\pi = \{b_1, \ldots, b_k\}$, $|c|$ refers to the number of assignments of c and $E_{c,i}$ denotes the ith assignment of c. Enumerating the satisfying assignments will now induce sets of simultaneously enabled commands and the corresponding target block combinations.

Parametric Markov chains. If the probabilities in a given probabilistic program are not concrete values but rather *parameters*, it corresponds to a parametric

Markov chain [8] instead of a DTMC. As the only part of our algorithm that deals with probabilities is the computation of the probability distribution induced by a command and a target block combination, it is fairly straightforward to incorporate parameters. Instead of computing a concrete value associated with each successor block, we symbolically derive an expression involving the parameters and only consider two parametric probability distributions equal if they syntactically coincide (in a certain normal form). This way, the computed quotient preserves all PCTL* properties for *all* possible parameter values.

5 Conclusion and Further Work

We have presented an SMT-based approach to extract, from a probabilistic program specified in the PRISM modelling language, the Markov chain representing its coarsest bisimulation quotient. No state space is generated—our bisimulation minimisation is a truly symbolic program manipulation. Experiments yielded encouraging results, even without optimisations such as formula simplification, which we plan to incorporate in future work. Application of the SMT-based approach to either parametric programs or programs with infinite state space but finite bisimulation quotient is straightforward and the approach can easily be adapted to perform compositional minimisation. We therefore believe that this approach represents a promising alternative to enumerative and BDD-based bisimulation minimisation algorithms.

Acknowledgements. The first two authors are funded by the EU FP7 project CARP (see http://www.carpproject.eu/) and the MEALS project, and the work was part-funded by the ERC Advanced Grant VERIWARE. We also thank Marta Kwiatkowska for facilitating the first author's visit to Oxford, where this work was initiated.

References

1. Aziz, A., Singhal, V., Balarin, F.: It Usually Works: The Temporal Logic of Stochastic Systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 155–165. Springer, Heidelberg (1995)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
3. Buchholz, P.: Exact and ordinary lumpability in finite Markov chains. Journal of Applied Probability 31, 59–75 (1994)
4. Coste, N., Garavel, H., Hermanns, H., Lang, F., Mateescu, R., Serwe, W.: Ten Years of Performance Evaluation for Concurrent Systems Using CADP. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 128–142. Springer, Heidelberg (2010)
5. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. Inf. Process. Lett. 87(6), 309–315 (2003)
6. Fisler, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. Formal Methods in System Design 21(1), 39–78 (2002)

7. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
8. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. STTT 13(1), 3–19 (2011)
9. 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)
10. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
11. Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perf. Ev. 68(2), 90–104 (2011)
12. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: Game-based probabilistic predicate abstraction in PRISM. Electr. Notes Theor. Comput. Sci. 220(3), 5–21 (2008)
13. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Formal Methods in System Design 36(3), 246–280 (2010)
14. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry Reduction for Probabilistic Model Checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
16. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
17. Mumme, M., Ciardo, G.: A Fully Symbolic Bisimulation Algorithm. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 218–230. Springer, Heidelberg (2011)
18. Valmari, A., Franceschinis, G.: Simple $O(m \log n)$ Time Markov Chain Lumping. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 38–52. Springer, Heidelberg (2010)
19. Wachter, B., Zhang, L., Hermanns, H.: Probabilistic model checking modulo theories. In: QEST, pp. 129–140 (2007)
20. Wimmer, R., Derisavi, S., Hermanns, H.: Symbolic partition refinement with automatic balancing of time and space. Perform. Eval. 67(9), 816–836 (2010)
21. 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)

Hybrid Automata-Based CEGAR for Rectangular Hybrid Systems

Pavithra Prabhakar[1], Parasara Sridhar Duggirala[2], Sayan Mitra[2], and Mahesh Viswanathan[2]

[1] IMDEA Software Institute, Madrid, Spain
[2] University of Illinois at Urbana-Champaign, Urbana, IL

Abstract. In this paper we present a framework for carrying out counter-example guided abstraction-refinement (CEGAR) for systems modelled as rectangular hybrid automata. The main difference, between our approach and previous proposals for CEGAR for hybrid automata, is that we consider the abstractions to be hybrid automata as well. We show that the CEGAR scheme is semi-complete for the class of rectangular hybrid automata and complete for the subclass of initialized rectangular automata. We have implemented the CEGAR based algorithm in a tool called HARE, that makes calls to HyTech to analyze the abstract models and validate the counterexamples. Our experiments demonstrate the usefulness of the approach.

1 Introduction

Direct model checking of realistic hybrid systems is in general undecidable and often foiled by the state-space explosion problem. Hence, one has to rely on some sort of abstraction. Finding the right abstraction is in itself a difficult problem. To this end, the *counterexample guided abstraction refinement* (CEGAR) [7] technique (see Section 3) which combines automatic refinement with model checking has gained preeminence in a number of contexts [4,19,8,14] including in timed and hybrid systems [2,6,5,24,11,23,9,20].

The space over which CEGAR performs the abstractions and refinements is key in determining both the efficiency (of model checking) and the completeness of the procedure. For example, in [2,6,5,24,23] abstraction-refinement is carried out in the space of finite-state discrete transition systems. Computing the transitions for this abstract finite state machine involves computing the unbounded time reachable states from the states in the concrete system corresponding to a particular state in the abstract system, which is difficult in practice for hybrid systems with complex continuous dynamics.

In this paper, we investigate CEGAR in the context of abstractions which are hybrid systems as well. When compared to using finite-state abstractions, using hybrid abstractions in a CEGAR framework requires carrying out computationally simpler tasks when constructing abstractions, refining them and validating counterexamples. Instead of the computationally expensive unbounded time successor computation, constructing hybrid abstractions only involves making local

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 48–67, 2013.

checks about flow equations. Moreover when validating counterexamples in a hybrid CEGAR scheme, one is only required to compute time-bounded successors (see 4.4). Computing time-bounded successors is often computationally easier than computing time-unbounded successors; for example, for automata with linear differential dynamics, time-bounded posts can be efficiently approximated, while no such algorithms exist for time-unbounded posts.

In this paper, we focus on *hybrid abstraction*-based CEGAR framework for rectangular hybrid systems. We abstract such automata using *initialized rectangular hybrid automata* [16]. The choice of initialized rectangular hybrid automata as the abstract space is motivated by the desire to have a rich space of abstractions, with a decidable model checking problem, and effective tools to analyze them. Our abstraction consists of the following operations: collapsing the control states and transitions, dropping the continuous variables and scaling the variables. Variable scaling changes the constants that appear in the abstract hybrid automaton which in turn can positively influence the efficiency of model checking the abstract automaton. Our refinement algorithm involves splitting control states/transitions, and/or adding variables that may have new dynamics (due to scaling).

Our main results in this paper are *complete/semi-complete* CEGAR algorithms for rectangular hybrid systems - (semi-completeness) if the hybrid system we are analyzing is a *rectangular* hybrid automaton and is faulty, then our CEGAR algorithm will terminate by demonstrating the error; (completeness) on the other hand, if the hybrid system is an *initialized rectangular* hybrid automaton then our CEGAR algorithm will *always* terminate. Such completeness results are usually difficult to obtain. In our case, one challenge is the fact that the collection of abstract counterexamples is not enumerable as the executions of an abstract hybrid system are uncountable. Thus, in order to argue that all abstract counterexamples are eventually considered, we need to change the notion of a counterexample to be a (infinite) set of executions, rather than a single execution. This change in the definition of counterexample also forces the validation algorithms to be different. The completeness proof then exploits topological properties, like compactness, to argue for termination.

Another highlight of our presentation is that we view our CEGAR framework as a composition of a few CEGAR loops. To carry this out, we identify some concepts and obtain a few results about the composition of CEGAR loops. Such a compositional approach simplifies the presentation of the refinement algorithm in our context, which is otherwise unwieldy, and helps identify more clearly the subtle concepts needed for the completeness proof to go through.

We have implemented our CEGAR based algorithm for rectangular hybrid automata in a tool that we call **H**ybrid **A**bstraction **R**efinement **E**ngine (HARE[1]). HARE makes calls to HyTech [17] to analyze abstract models and generate counterexamples; we considered PHAVer [13] and SpaceEx [12], but at the time of writing they do not produce counterexamples. We analyzed the tool on several benchmark examples which illustrate that its total running time is comparable with that of HyTech, and on some examples HARE is a couple of orders of

magnitude faster. Moreover, in some cases HARE can prove safety with dramatically small abstractions. Fair running-time comparison of HARE with other tools, such as d/dt [3] and checkmate [5], is not possible because of the differences in the runtime environments. Experimental comparison of finite-state and hybrid abstractions was also not possible because to the best of our knowledge, the tools implementing finite-state abstractions are not publicly available. We believe that in terms of efficiency, the approaches of finite state abstractions and hybrid abstractions are incomparable.

Related Work. CEGAR with hybrid abstractions have been investigated in [9,20] where the abstractions are constructed by ignoring certain variables. Counterexamples are used to identify new variables to be added to the abstraction. This approach has been carried out for timed automata [9] and linear hybrid automata [20]. In comparison to the above, our abstractions (may) change both the control graph and variable dynamics, and are not restricted to only forgetting continuous variables. In contrast, the scheme in [20] considers a more general class of hybrid automata, though the abstractions in that scheme are not progressively refined. Finally, in [10] hybrid systems with flows described by linear differential equations are approximated by rectangular hybrid automata. Even though, their scheme progressively refines abstractions, the refinements are not guided by counter-examples.

2 Preliminaries

Notation, Images and Inverse Images. Let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{R}_{\geq 0}$ denote the set of natural numbers, integers, rationals, reals and non-negative reals, respectively. Given a function $f : A \to B$ and a subset $A' \subseteq A$, $f(A')$ is defined to be the set $\{f(x) \mid x \in A'\}$. Similarly, for $B' \subseteq B$, $f^{-1}(B')$ is the set $\{x \mid \exists y \in B', f(x) = y\}$. When B' is a singleton set $\{y\}$, we also use $f^{-1}(y)$ to denote $f^{-1}(\{y\})$.

Transition Systems. A *transition system* $\mathcal{T}$ is a tuple $(S, S^0, \Sigma, \longrightarrow)$, where S is a set of states, $S^0 \subseteq S$ is a set of initial states, Σ is a set of transition labels, and $\longrightarrow \subseteq S \times \Sigma \times S$ is a transition relation. We call $(s, a, s') \in \longrightarrow$ a transition of $\mathcal{T}$ and denote it as $s \xrightarrow{a} s'$. We denote the elements of a transition system using appropriate subscripts. For example, the set of states of a transition system $\mathcal{T}_i$ is denoted by S_i.

An *execution fragment* σ of a transition system $\mathcal{T}$ is a sequence of transitions $t_0 t_1 t_2 t_3 \cdots t_n$, such that $s'_i = s_{i+1}$ for $0 \leq i < n$, where t_i is given by $s_i \xrightarrow{a_i} s'_i$. We denote the above execution fragment by $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_{n-1} \xrightarrow{a_{n-1}} s_n$. We say that the length of σ is n, denoted by $|\sigma|$. An *execution* of σ is an execution fragment starting from a state in S^0. We denote the set of all execution fragments of $\mathcal{T}$ by $ExecF(\mathcal{T})$ and the set of all executions by $Exec(\mathcal{T})$.

Given a set of states $S' \subseteq S$ and a symbol $a \in \Sigma$, $Pre_{\mathcal{T}}(S', a)$ is defined to be the set $\{s_1 \mid \exists s_2 \in S' : s_1 \xrightarrow{a} s_2\}$ and $Post_{\mathcal{T}}(S', a)$ as $\{s_2 \mid \exists s_1 \in S' : s_1 \xrightarrow{a} s_2\}$. Given a subset Σ' of Σ, $Pre_{\mathcal{T}}(S', \Sigma') = \bigcup_{a \in \Sigma'} Pre_{\mathcal{T}}(S', a)$ and $Post_{\mathcal{T}}(S', \Sigma') = \bigcup_{a \in \Sigma'} Post_{\mathcal{T}}(S', a)$.

Hybrid Automata. A hybrid system is a system which exhibits mixed discrete-continuous behaviors. A popular model for representing hybrid systems is that of hybrid automata [16], which combine finite state automata modeling the discrete dynamics, and differential equations or inclusions modeling the continuous dynamics. An execution of such a system begins in a state of the automaton with some values to the variables representing the continuous dynamics. It then either evolves continuously, wherein only the values of the continuous variables change according to the continuous dynamics associated with the current discrete state of the automaton, or takes a discrete transition, where potentially both the discrete and continuous state of the automaton can change. During the latter, the state of the automaton changes from the source of an edge to its target, and the discrete transition is enabled only if the continuous state before the transition satisfies the enabling condition associated with the edge. The value of the continuous state after the transition either remains the same or is reset to some other value.

Definition 1. *A* hybrid automaton $\mathcal{H}$ *is a tuple* (*Loc*, *Edges*, *Source, Target,* q^{init}, n, $Cont_0$, *inv*, *flow*, *jump*), *where*

- *Loc is a finite set of* (discrete) control states *or* locations.
- *Edges is a finite set of* edges.
- *Source, Target: Edges* $\rightarrow$ *Loc are functions which associate a source and a target location to every edge, respectively.*
- $q^{init} \in$ *Loc is the* initial location. *The components above represent the discrete part of the automaton.*
- $n \in \mathbb{N}$ *is the dimension of* $\mathcal{H}$, *also denoted by* $Dim(\mathcal{H})$, *which represents the number of continuous variables in the system. The set of continuous states is given by* $Cont = \mathbb{R}^n$.
- $Cont_0 \subseteq Cont$ *is the* initial set of continuous states.
- *inv:* $Loc \rightarrow 2^{Cont}$ *associates with every location an* invariant *set. The continuous state of the system belongs to the invariant of a location as long as the control is in that location.*
- *flow:* $Loc \rightarrow 2^{Traj(Cont)}$ *associates with every location a set of trajectories, where* $Traj(Cont)$ *is the set of continuous functions from* $\mathbb{R}_{\geq 0} \rightarrow Cont$.
- *jump:* $Edges \rightarrow 2^{Cont \times Cont}$, *associates with every edge a set of pair of states describing the value of the continuous state before and after the edge is taken.*

Next we present the semantics of a hybrid automaton as a transition system it represents. The semantics of a hybrid automaton $\mathcal{H}$ is defined in terms of the transition system $[\![\mathcal{H}]\!] = (Q, Q^0, \Sigma, \longrightarrow)$ over $\Sigma = \mathbb{R}_{\geq 0} \cup Edges$, where $Q = Loc \times Cont$, $Q^0 = q^{init} \times Cont_0$, and the transition relation $\longrightarrow$ is given by:

- *Continuous transitions* - For $t \in \mathbb{R}_{\geq 0}$, $(q_1, x_1) \xrightarrow{t} (q_2, x_2)$ iff $q_1 = q_2 = q$ and there exists a function $f \in flow(q)$ such that $x_1 = f(0)$, $x_2 = f(t)$ and for all $t' \in [0, t]$, $f(t') \in inv(q)$.
- *Discrete transitions* - For $e \in Edges$, $(q_1, x_1) \xrightarrow{e} (q_2, x_2)$ iff $q_1 = Source(e)$, $q_2 = Target(e)$, $x_1 \in inv(q_1)$, $x_2 \in inv(q_2)$ and $(x_1, x_2) \in jump(e)$.

We focus on the problem of *control state reachability*, namely, given a hybrid automaton $\mathcal{H}$ and a location $q \neq q^{init}$, is q reachable in $\mathcal{H}$, or equivalently does there exist an execution of $\mathcal{H}$ from a state in $\{q^{init}\} \times Cont_0$ to $\{q\} \times Cont$? Typically, q is a "bad" or "unsafe" location that we do not want to reach, and we are interested in determining the safety of the system, namely, no execution reaches the unsafe location.

3 CEGAR Framework

A counter-example guided abstraction refinement algorithm consists of the four steps, namely, *abstraction*, *model-checking*, *validation* and *refinement*. We focus on safety verification here. CEGAR loop begins with the construction of an abstraction (an overapproximation) of the original system (also called the concrete system). The abstract system is then model-checked to determine if there exists an execution from the initial location to an unsafe location. Such a path if one exists is called an *abstract counter-example*. If the abstract system has no counter-examples, then it can be deduced from the properties of abstraction that even the concrete system does not have any counter-examples, and hence is safe. However, if an abstract counter-example is returned in the model-checking phase, then one cannot in general make any conclusions about the safety of the concrete system, and the counterexample is validated to determine if there exist a counter-example in the concrete system corresponding to it. If a concrete counter-example is found, then the concrete system is unsafe, and the concrete counter-example exhibits a bug in the system. Otherwise, the analysis in validating the abstract counter-example is used to construct a new abstract system which is a refinement of the current abstract system. The CEGAR algorithm continues with the model-checking of the new abstract system. In general, the CEGAR loop might not terminate.

The purpose of this section is to fix notation and highlight some differences with the standard CEGAR loop. We present several CEGAR algorithms in the next section, which provide various guarantees about the termination, namely, completeness and semi-completeness. Completeness refers to the fact that there are only finitely many iterations of the CEGAR loop in any execution; and semi-completeness refers to the fact that the CEGAR loop always terminates on a faulty machine exhibiting a counter-example. Semi-completeness is easy to guarantee when the set of concrete executions of a system are (efficiently) enumerable, which is lacking for the class of systems we consider. Hence, we need to change the notion of a counter-example to encapsulate a possibly (uncountable) number of counter-examples, and perform the validation simultaneously on this infinite set. This requires us to perform validation in a slightly different manner, namely, we first need to find the actual set of abstract executions corresponding to the counter-example, and then perform the standard validation on the computed set. Further, in order to guarantee termination, we need to at the least guarantee that we make progress in each iteration of the CEGAR loop.

In the rest of the section, we setup notation and explain our notion of counter-example, the modified validation algorithm, and distill some local condition which ensure progress of the CEGAR loop.

We fix some notation for the rest of this section. Our concrete systems and abstract systems are both hybrid automata. Let $\mathcal{H}_C$ be a concrete hybrid automaton and let $\mathcal{H}_A$ be an abstract hybrid automaton. Let $\mathcal{T}_C = [\![\mathcal{H}_C]\!] = (S_C, S_C^0, \Sigma_C, \longrightarrow_C)$ and $\mathcal{T}_A = [\![\mathcal{H}_A]\!] = (S_A, S_A^0, \Sigma_A, \longrightarrow_A)$ be the transition systems associated with $\mathcal{H}_C$ and $\mathcal{H}_A$, respectively. Let us fix an unsafe location $q_C^{unsafe} \neq q_C^{init}$ in $\mathcal{H}_C$, and we want to check if q_C^{unsafe} is reachable in $\mathcal{H}_C$.

3.1 Abstraction

In the next section, we present several methods for constructing abstractions. Here, we define formally the relation that holds between a system and its abstraction.

Given transition systems $\mathcal{T}_1 = (S_1, S_1^0, \Sigma_1, \longrightarrow_1)$ and $\mathcal{T}_2 = (S_2, S_2^0, \Sigma_2, \longrightarrow_2)$, an *abstraction* or *simulation* function from $\mathcal{T}_1$ to $\mathcal{T}_2$ is a pair of functions $\alpha = (\alpha_S, \alpha_\Sigma)$, where $\alpha_S : S_1 \to S_2$ and $\alpha_\Sigma : \Sigma_1 \to \Sigma_2$ such that

- $\alpha_S(S_1^0) \subseteq S_2^0$, and
- for every $s_1, s_1' \in S_1$ and $a_1 \in \Sigma_1$, $s_1 \xrightarrow{a_1}_1 s_1'$ implies $\alpha_S(s_1) \xrightarrow{\alpha_\Sigma(a_1)}_2 \alpha_S(s_1')$.

We say that $\mathcal{T}_2$ is an abstraction of $\mathcal{T}_1$, denoted by $\mathcal{T}_1 \preceq \mathcal{T}_2$. We denote the fact that α is an abstraction function from $\mathcal{T}_1$ to $\mathcal{T}_2$ by $\mathcal{T}_1 \preceq_\alpha \mathcal{T}_2$.

Notation. Given an abstraction function $\alpha = (\alpha_S, \alpha_\Sigma)$, we will drop the subscripts S and Σ when it is clear from the context. For example, for a state s, we will use $\alpha(s)$ to mean $\alpha_S(s)$. Note that if α is an abstraction function from $\mathcal{T}_1$ to $\mathcal{T}_2$ and $\sigma = s_1 \xrightarrow{a_1}_1 \cdots s_n$ is an execution fragment of $\mathcal{T}_1$, $\sigma' = \alpha(s_1) \xrightarrow{\alpha(a_1)}_2 \cdots \alpha(s_n)$ is an execution fragment of $\mathcal{T}_2$. Let us denote $\alpha(s_1) \xrightarrow{\alpha(a_1)}_2 \cdots \alpha(s_n)$ by $\alpha(\sigma)$.

Let us fix an abstraction function α from the concrete system $\mathcal{T}_C$ to the abstract system $\mathcal{T}_A$. Since, we are interested in control state reachability, we need certain consistency conditions on α to ensure that property is preserved. We assume that there exists a location $q_A^{unsafe} \neq q_A^{init}$ in $\mathcal{H}_A$ satisfying $\alpha(\{q_C^{unsafe}\} \times Cont_C) \subseteq \{q_A^{unsafe}\} \times Cont_A$, that is, α maps the elements of the unsafe set of concrete states to states corresponding to an unique location of $\mathcal{H}_A$. Let $Unsafe_C = \{s_C\} \times Cont_C$ and $Unsafe_A = \{s_A\} \times Cont_A$. Note that if q_A^{unsafe} is not reachable in the abstract hybrid automaton $\mathcal{H}_A$, then q_C^{unsafe} is not reachable in the concrete hybrid automaton $\mathcal{H}_C$.

3.2 Counter-Examples

If q_A^{unsafe} is reachable in $\mathcal{H}_A$, then the model-checker returns a counter-example. In order to guarantee semi-completeness, we need a set of counter-examples which are enumerable, and spawn the set of all executions of the system. Hence,

we define a counter-example to be a path in the control flow graph of the abstract hybrid automaton, which is feasible, that is, has an execution corresponding to it. Note that such a counter-example exhibits potentially an infinite set of unsafe abstract executions.

Given an element $\pi = q_0 e_0 q_1 \cdots q_n \in (LocEdges)^* Loc$, define $PathToExecF(\pi)$ to be the set of all execution $\sigma = (q_0, x_0) \xrightarrow{t_0} (q_0, y_0) \xrightarrow{e_0} (q_1, x_1) \xrightarrow{t_1} (q_1, y_1) \cdots (q_{n-1}, y_{n-1}) \xrightarrow{e_{n-1}} (q_n, x_n) \xrightarrow{t_n} (q_n, y_n)$. Formally, a *counter-example* of a hybrid automaton $\mathcal{H}$ given an unsafe location q^{unsafe} is an alternating sequence of locations and edges, that is, an element π of $q^{init} Edges(LocEdges)^* \; q^{unsafe}$ such that $PathToExecF(\pi)$ is not empty. The length of a counter-example is the number of elements in the sequence. We will call a counter-example of $\mathcal{H}_C$ with unsafe location q_C^{unsafe}, a *concrete counter-example*, and a counter-example of $\mathcal{H}_A$ with unsafe location q_C^{unsafe}, an *abstract counter-example*.

3.3 Validation

We think of an abstract counter-example as representing a possibly infinite set of abstract unsafe executions. Hence, validation needs to check if there is a concrete unsafe execution corresponding to any of the abstract unsafe executions represented by the abstract counter-example. This requires us to perform validation in a slightly different manner. Validation takes place in two phases: In the first phase, a forward analysis is done to compute a representation of the abstract executions corresponding to the counter-example. In particular, the set of abstract states reached by traversing the abstract counter-example is computed. In the next phase, a backward reachability computation is performed in the concrete system, along the potentially infinite set of abstract executions computed in the previous step, and represented by a sequence of sets of abstract states. The precise algorithm is given in Figure 1. There exists a concrete unsafe execution corresponding to the abstract counter-example π_A iff $Reach_{\pi_A,\alpha}(0) \cap S_C^0 \neq \emptyset$.

Remark 1. Observe that just running a standard backward reachability algorithm on the counter-example does not suffice, since the reach sets thus obtained might contain concrete state which do not correspond to an actual abstract execution. This is because, running a backward reachability would correspond to "validating" all abstract execution fragments corresponding to any "subpath" of the counter-example, simultaneously.

Given an execution fragment σ' of $\mathcal{T}_A$ and an abstraction function α from $\mathcal{T}_C$ to $\mathcal{T}_A$, we denote by $Conc_\alpha(\sigma')$, the set of execution fragments in $\mathcal{T}_C$ corresponding to $\mathcal{T}_A$, namely, the set $\{\sigma \in ExecF(\mathcal{T}_C) \,|\, \alpha(\sigma) = \sigma'\}$. Validation is the process of checking if $Conc_\alpha(PathToExecF(\pi_A))$ contains an execution of $\mathcal{T}_C$ reaching s_C.

Proposition 1. *$Conc_\alpha(PathToExecF(\pi_A))$ contains an execution of $\mathcal{T}_C$ reaching s_C iff $Reach_{\pi_A,\alpha}(0) \cap S_C^0 \neq \emptyset$.*

If $Reach_{\pi_A,\alpha}(0) \cap S_C^0 = \emptyset$, then we call π_A a *spurious* counter-example.

Input: π_A, an abstract counter-example in $\mathcal{H}_A$ of length l.

Phase 1: Forward reachability in the abstract automaton.

Compute $FReach_{\pi_A}(i)$, for $0 \leq i \leq 2l+1$.
- $FReach_{\pi_A}(0) = S^0_A$.
- $FReach_{\pi_A}(i+1) = Post_{\mathcal{T}_A}(FReach_{\pi_A}(i), a)$, where
  - (Time Elapse) $a = \mathbb{R}_{\geq 0}$ if i is even, and
  - (Edge) $e'_{(i-1)/2}$ if i is odd, for $0 \leq i < 2l+1$.

Phase 2: Backward reachability in the concrete automaton.

Compute S_k for $0 \leq i \leq 2l+1$:

$$S_k = \alpha^{-1}(FReach_{\pi_A}(k)), 0 \leq i < 2l+1$$

$$S_{2l+1} = \alpha^{-1}(FReach_{\pi_A}(k)) \cap (q_C^{unsafe} \times Cont_C)$$

Compute $Reach_{\pi_A,\alpha}(k)$, $0 \leq k \leq 2l+1$.
- $Reach_{\pi_A,\alpha}(2l+1) = S_{2l+1}$.
- $Reach_{\sigma_A,\alpha}(k) = S_k \cap Pre_{\mathcal{T}_C}(Reach_{\pi_A,\alpha}(k+1), a)$, where
  - (Time elapse) $a = \mathbb{R}_{\geq 0}$ if k is even, and
  - (Edge) $\alpha^{-1}(e'_{(k-1)/2})$, if k is odd.

Fig. 1. Validation Algorithm

3.4 Refinement

We formalize the conditions which ensure that the refinement is making progress by "eliminating" spurious abstract counter-examples. The refinement algorithms we present in the next section ensure the progress conditions presented here.

Definition 2. *Given two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{T}_1 \preceq \mathcal{T}_2$, a transition system $\mathcal{T}_3$ is said to be a* refinement *of $\mathcal{T}_2$ with respect to $\mathcal{T}_1$, if $\mathcal{T}_1 \preceq \mathcal{T}_3 \preceq \mathcal{T}_2$.*

Notation. We will say $\mathcal{H}_3$ is a refinement of $\mathcal{H}_2$ with respect to $\mathcal{H}_1$ to mean that $[\![\mathcal{H}_3]\!]$ is a refinement of $[\![\mathcal{H}_2]\!]$ with respect to $[\![\mathcal{H}_1]\!]$, and denote it by $\mathcal{H}_1 \preceq \mathcal{H}_3 \preceq \mathcal{H}_2$.

Our goal is to find a refinement $\mathcal{H}_R$ such that $\mathcal{H}_C \preceq \mathcal{H}_R \preceq \mathcal{H}_A$. Note that $\mathcal{H}_R = \mathcal{H}_A$ is such a system, however, we want to make progress by eliminating the spurious counter-example. Hence, we define good refinements to be those in which some "potential" execution fragment of the counter-example in the current abstraction is not a "potential" execution fragment of any counter-example in the refinement.

To formalize progress, we need some definitions. Given a transition system $\mathcal{T}$, a *potential execution fragment* of $\mathcal{T}$ is a sequence $\rho = s_0 a_0 s_1 a_1 \cdots s_{l-1} a_{l-1} s_l$ alternating between elements of S and Σ. Further, given an abstraction function γ from $\mathcal{T}_1$ to $\mathcal{T}_2$, and an execution fragment $\sigma' = s'_0 \xrightarrow{a'_0}_2 s'_1 \xrightarrow{a'_1}_2 \cdots s'_{l-1} \xrightarrow{a'_{l-1}}_2 s'_l$ of $\mathcal{T}_2$, $Potential_\gamma(\sigma')$ is the set of all potential execution fragments $\rho = s_0 a_0 s_1 a_1 \cdots s_{l-1} a_{l-1} s_l$ of $\mathcal{T}_1$ such that $\gamma(s_i) = s'_i$ and $\gamma(a_i) = a'_i$. Note that a potential execution fragment might not correspond to an actual execution fragment.

Definition 3. *Let $\mathcal{H}_C \preceq_\alpha \mathcal{H}_A$ and $\mathcal{H}_C \preceq_\beta \mathcal{H}_R$. Given a spurious counter-example π_A of $\mathcal{H}_A$, a refinement $\mathcal{H}_R$ of $\mathcal{H}_A$ with respect to $\mathcal{H}_C$ is said to be* good *with respect to π_A if there exists a $\rho \in Potential_\alpha(PathToExecF(\pi_A))$ such that $\rho \notin Potential_\beta(PathToExecF(\pi_R))$ for any counter-example π_R of $\mathcal{H}_R$.*

In validating a spurious counter-example π_A of $\mathcal{H}_A$, we see that $Reach_{\pi_A,\alpha}(0) \cap S^0_C$ is empty. Note that if $Reach_{\pi_A,\alpha}(k) = \emptyset$ for some k, then $Reach_{\pi_A,\alpha}(i)$

is empty for all $i \leq k$. Let $\hat{k}$ be the largest integer such that $Reach_{\pi_A,\alpha}(\hat{k})$ is empty, if $Reach\pi_A, \alpha(k)$ is empty for some k, otherwise, let $\hat{k} = 0$ (since $Reach_{\pi_A,\alpha}(0) \cap S_C^0$ is definitely empty). We call $\hat{k}$ the *infeasibility index* of π_A with respect to α. The next proposition states a local sufficient condition to ensure that a refinement is good.

Proposition 2. *Let α be the an abstraction function which is surjective. Let $\pi_A = s'_0 e'_0 s'_1 \cdots s'_{l-1} e'_{l-1} s'_l$ be a spurious counter-example of $\mathcal{H}_A$ with an infeasibility index $\hat{k}$ with respect to α. Suppose that $\mathcal{H}_C \preceq_\beta \mathcal{H}_R$ is a refinement of $\mathcal{H}_A$ with respect to $\mathcal{H}_C$ satisfying:*

$$Post_{[\![\mathcal{H}_R]\!]}(\beta(S_{\hat{k}}), \beta(a_{\hat{k}})) \cap \beta(Reach_{\pi_A,\alpha}(\hat{k}+1)) = \emptyset, \tag{1}$$

where $S_{\hat{k}}$ is as defined in Figure 7 and $a_{\hat{k}} = \mathbb{R}_{\geq 0}$ if $\hat{k}$ is even and is $\{\alpha^{-1}(e'_{(\hat{k}-1)/2})\}$ otherwise. Then $\mathcal{H}_R$ is a refinement of $\mathcal{H}_A$ with respect to $\mathcal{H}_C$ which is good with respect to π_A.

Remark 2. Validation can be done by checking if $Conc_\alpha(PathToExecF(\pi_A))$ contains an execution of $\mathcal{T}_C$ reaching s_C, which can be verified by performing a backward reachability in the concrete system with respect to the abstract counter-example and checking if the reach set becomes empty. However, in order to guarantee progress in the refinement step, in particular, to be able to perform refinement which satisfies Equation 1, we need to identify and eliminate a potential concrete execution of some abstract unsafe execution.

3.5 Completeness, Semi-completeness and Composition

A CEGAR algorithm for a class of systems takes as input a finite representation of a system from the class and (a) either outputs "YES" or (b) outputs "NO" and returns a counter-example of the system, or (c) does not terminate. If it outputs "YES", then it is guaranteed that the system is safe, that is, the unsafe location is not reachable, and if it outputs "NO", then it is unsafe. We will assume that the different phases of any loop of the CEGAR algorithm terminate, but CEGAR algorithm itself may or may not terminate. Next, we define the notions of completeness and semi-completeness of a CEGAR algorithm.

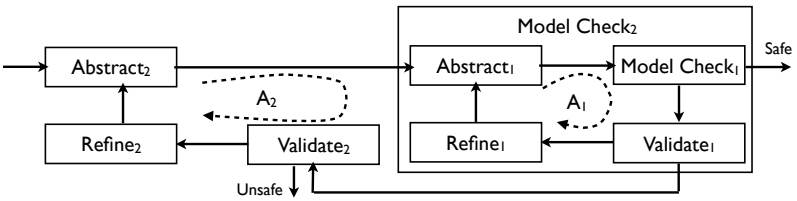


Fig. 2. $A_2[A_1]$: Composition of CEGAR Algorithm A1 with A2

Definition 4. *A CEGAR algorithm is said to be* complete *for a subclass C of hybrid automata if it terminates for all inputs from the class C. It is said to be* semi-complete *if it terminates at the least for inputs from C which are unsafe, that is, have an execution to an unsafe location.*

We say that a CEGAR algorithm is *fair*, if it returns a smallest length counter-example whenever it terminates on an unsafe system, under the assumption that the model-checker always returns a smallest length counter-example.

Note that a CEGAR algorithm is essentially a model-checking algorithm. Hence, we can compose CEGAR algorithms by using a CEGAR algorithm A_1 as a model-checker for a CEGAR algorithm A_2 as shown in Figure 2. We denote the composed algorithm by $A_2[A_1]$.

Proposition 3. *Let A_i be a CEGAR algorithm with input and abstraction spaces C_i and D_i respectively, for $i = 1, 2$, such that $D_2 \subseteq C_1$. Then:*

- *If A_1 and A_2 are complete and fair CEGAR algorithms, then $A_2[A_1]$ is complete and fair.*
- *If A_2 is semi-complete and fair, and if A_1 is complete and fair, then $A_2[A_1]$ is also semi-complete and fair.*

4 CEGAR for Rectangular Hybrid Automata

In this section, we focus on a subclass of hybrid automata called rectangular hybrid automata. We present three CEGAR algorithms for this class. In order to keep the presentation simple, we choose to illustrate the ideas in the algorithms using examples. The formal description and details can be found in [22].

We begin with a brief overview of the class of rectangular hybrid systems. A rectangular (hybrid) automaton is a hybrid automaton in which the invariants, guards, jumps and flow are specified using rectangular constraints. A rectangular constraint is of the form $x \in I$, where x is a variable and I is an interval whose finite end-points are integers. Figure 3 shows a rectangular hybrid automaton.

It has four locations, namely, l_1, l_2, l_3 and l_4, as shown by the circles and four edges e_1, e_2, e_3 and e_4, as shown by the arrows between the circles. The invariant at location l_3 is given by $x \in [-1, 1]$. To keep the diagram simple, we have omitted the constraints $x \in [-10, 10]$ and $y \in [-10, 10]$ from the invariants of every location. The flow is specified by rectangular differential inclusions of the form $\dot{x} \in I$. The flow associated with it are all functions whose projection to the x-component is such that the derivative with respect to time belongs to the interval I at all times.

The jump relation is specified using two kinds of constraints, namely, guards and resets. A guard specifies the enabling condition on the edge, and the reset specifies the value of the continuous state after the edge is taken. An edge is labelled by a constraint of the form $x \in I$, which specifies that the edge can be taken when the value of x belongs to the interval I. If an edge is labelled by a constraint of the form $x :\in I$, it means that the value of x after the edge

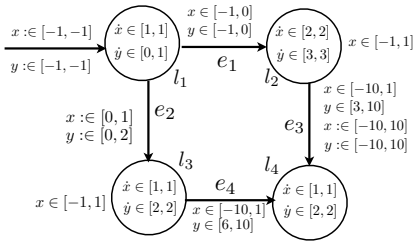


Fig. 3. $\mathcal{H}_1$: An example of a rectangular hybrid automaton

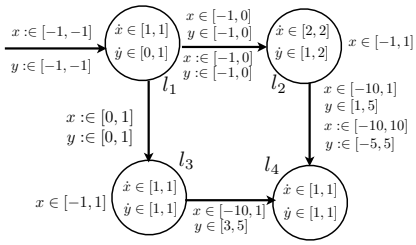


Fig. 4. $\mathcal{H}_5$: Flow abstraction of $\mathcal{H}_2$ scaling down y by a factor of 2

is taken is reset non-deterministically to some value in the interval I. When a constraint of the form $x :\in I$ is absent, it means that the value of a variable remains the same after taking an edge. We call a reset of the first form as *strong reset* and a reset of the second form as an *identity reset*. Note that the jump relation associated with a constraint $x \in I_1$, $x :\in I_2$ is $I_1 \times I_2$, where as that associated with just $x \in I$ is $\{(x, x) \mid x \in I\}$.

The control state reachability problem is undecidable for the class of rectangular hybrid automata [15]. Hence, to ensure that the model-checking phase of a CEGAR loop terminates, we consider as the abstraction space, a subclass of rectangular hybrid automata called *initialized rectangular hybrid automata*, which have the property that whenever the differential inclusions associated with a variable is different for the source and target of an edge, then the edge is necessarily labelled by a reset constraint for that variable, that is, the value of the variable is non-deterministically reset to some value in an interval (as opposed to carrying over the value from previous location). The control state reachability problem has been shown to be decidable for the class of initialized rectangular hybrid automata [15]. For example, the variable x is non-initialized along the edge e_1, since the constraints associated with $\dot{x}$ in locations l_1 and l_2 are different, but the edge e_1 does not have a reset constraint for x.

Before presenting the CEGAR algorithm, let us discuss briefly the computability of the validation step. For a transition system $\mathcal{T}$ arising from a rectangular hybrid automaton $\mathcal{H}$, one can compute $Pre_{\mathcal{T}}(S, a)$ and $Post_{\mathcal{T}}(S, a)$ where S is any linear set and a is either the set of non-negative reals $\mathbb{R}_{\geq 0}$ or a subset of edges *Edges* of the hybrid automaton $\mathcal{H}$. Further, the resulting sets are linear too. This implies that $FReach_{\pi}(i)$ can be computed for any counter-example π of $\mathcal{H}$ and any position i in π. Further $FReach_{\pi}(i)$ is a linear set. We distill the conditions for $Reach_{\pi_A,\alpha}(i)$ to be computable in the following proposition.

Proposition 4. *Let α be an abstraction function from $\mathcal{H}_C$ to $\mathcal{H}_A$. Suppose that for any linear subset S of the state space, $\alpha^{-1}(S)$ is a linear set and can be computed. Then $Reach_{\pi_A,\alpha}(k)$ is a linear set and is computable for each k. Further, $Reach_{\pi_A,\alpha}(k)$ is a compact set.*

Proof. Compactness follows from the fact that the invariants associated with the locations are compact (closed and bounded).

In this section, we present three CEGAR algorithms. For each of these we present the "hybrid abstraction" which can be thought of as a symbolic representation of the abstraction function, and a specific method to construct an abstract system using the hybrid abstraction. Further, the resulting abstraction function will be such that it satisfies the hypothesis of Proposition 4. Hence, we can effectively carry out the validation phase. We will present our refinement algorithm which ensures progress as given by Equation 1.

The first CEGAR algorithm abstracts a rectangular hybrid automaton to an initialized rectangular hybrid automaton by abstracting the identity resets of the edges which violate the initialization condition by strong resets. This algorithm is semi-complete for the class of rectangular automata. Next, we present two CEGAR algorithms for the class of initialized rectangular automata, which are complete for this class. One abstracts a system, by merging together different locations/edges and the other abstracts by dropping/scaling variables. All these algorithms are fair. Hence, they can be composed as sketched in Proposition 3 to obtain more sophisticated CEGAR algorithms which are complete and semi-complete, respectively.

4.1 Strong Reset Abstraction Based CEGAR

Abstraction. The broad idea is to abstract a rectangular hybrid automaton to an initialized rectangular hybrid automaton by abstracting an identity reset which violates the initialization condition by a strong reset. A naive approach is to replace a constraint $x \in I$ associated with a pair non-initialized edge e and variable x, by the constraint $x \in I, x :\in I$. It transforms an identity reset to a strong reset, and is such that the jump relation associated with the new constraint $\{(v_1, v_2) \mid v_1, v_2 \in I\}$ is a superset of the jump relation before the transformation $\{(v, v) \mid v \in I\}$. Observe that one can interpret the identity reset $\{(v, v) \mid v \in I\}$ as an infinite set of strong resets $(\{(v, v)\})_{v \in I}$. We choose to abstract an identity reset by a finite set of strong resets for the verification to be computationally feasible. More generally, we abstract a constraint $x \in I$ by a set of constraints $x \in J_i, x :\in J_i, i \in K$, where $J_i, i \in K$ is a finite partition of I.

Consider the rectangular automaton $\mathcal{H}_1$ in Figure 3. The only non-initialized edge in the automaton is e_1. The strong rest abstraction of $\mathcal{H}_1$ is exactly the same as $\mathcal{H}_1$, except that the constraint associated with edge e_1 is $x \in [-1, 0]$, $x :\in [-1, 0]$, $y \in [-1, 0], y :\in [-1, 0]$. Let us call this automaton $\mathcal{H}_2$.

Refinement. The refinement step constructs a new abstraction by replacing the jump relation associated with a non-initialized edge by a smaller set. The validation step identifies the two sets as given by Equation 1 that need to be separated. One can show that the infeasibility index always corresponds to a non-initialized edge of the concrete automaton. The refinement corresponds to

refining the partition used in transforming the non-initialized edge to an initialized edge.

Let us consider the counter-example $l_1e_1l_2e_3l_4$ of the strong reset abstraction of $\mathcal{H}_1$, namely, $\mathcal{H}_2$, where l_4 is the unsafe state. Validation step returns that the sets $A = \{(-1, 0)\}$ and $B = \{(v_1, v_2) \mid v_1 \in [-1, 0], v_2 \in [-1, 0], v_1 \geq v_2\}$. The minimum distance (Euclidean) between A and B is $\sqrt{2}/2$. Hence, any square whose sides are $1/2$ units will not overlap with both A and B. Therefore, we partition the guard $x \in [-1, 0]$, $y \in [-1, 0]$ into square chunks of width $1/2$. So the edge e_1 in the refinement is labelled by the multi-rectangular constraints corresponding to the partition $(x \in [-1, -1/2], y \in [-1, -1/2])$, $(x \in [-1/2, 0], y \in [-1, -1/2])$, $(x \in [-1, -1/2], y \in [-1/2, 0])$, and $(x \in [-1/2, 0], y \in [-1/2, 0])$. Note that the refinement step could force us to use rational end-points, even if we begin with integer end-points for all the intervals which appear in the constraints.

Remark 3. Compactness of the reach set is a crucial property we exploit here, since this guarantees that there exists a minimum distance between the two sets that need to be separated according to Equation 1.

It is easy to see that if the model-checker always return a smallest counter-example in the abstract system when one exists, then the CEGAR algorithm will find a smallest length counter-example if it terminates. However, the CEGAR loop might not terminate in general (a consequence of the undecidability of control state reachability for rectangular automata). Hence, we obtain the following partial guarantee about the termination of the CEGAR loop.

Theorem 1. *The strong reset abstraction based CEGAR algorithm is semi-complete for the class of rectangular hybrid automata, and is fair.*

Remark 4. Note that the semi-completeness depends crucially on our choice of the notion of a counter-example. For example, the above theorem would not hold had we chose an abstract execution fragment as the notion of counter-example.

4.2 Control Abstraction Based CEGAR

We present a CEGAR algorithm for the class of initialized rectangular automata. Hence, this CEGAR loop can be used as a model-checker for the strong reset abstraction based CEGAR algorithm of the previous section.

Abstraction. As the name suggests, we abstract the underlying control flow graph of the initialized rectangular automaton. More precisely, we define a consistent partition of locations/edges and merge the locations/edges in the each of the partitions. We define the constraints for the invariants, differential inclusions, guards and resets to be a the smallest rectangular constraints which contain the corresponding constraints for the elements in each partition.

Figure 5 shows an abstraction of the initialized rectangular automaton $\mathcal{H}_2$ in which location l_2 and l_3 are merged and edges e_3 and e_4 are merged.

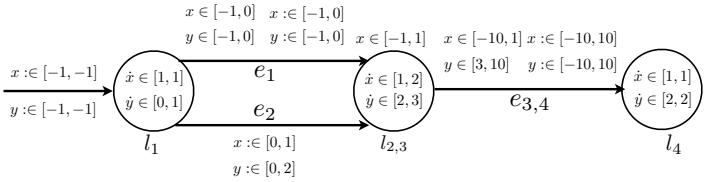


Fig. 5. $\mathcal{H}_3$: Control Abstraction of the Hybrid Automaton $\mathcal{H}_2$

The constraint on $\dot{x}$ in $l_{2,3}$ is a constraint which corresponds to the smallest rectangular set containing $[2,2]$ and $[1,1]$, the constraints on $\dot{x}$ in l_2 and l_3, respectively. Similarly, the guard for the variable y on the edge $e_{3,4}$ is the union of the sets $[3,10]$ and $[6,10]$.

We call the smallest rectangular set containing a given set, the *rectangular hull* of the set. We use the following rules in constructing the constraints for an abstract edge E. If all the concrete edges corresponding to E have identity resets, then E is an identity reset and the constraint for the guard is the rectangular hull of the constraints on the concrete edges. If at least one the edges does not have an identity reset, then the edges with identity reset are transformed into the naive strong reset explained in the previous section. Then the guard and reset are obtained by taking the rectangular hull of the the guards and resets, respectively, of the concrete strong reset transformed edges. The only exception to the above rule is when E is a singleton edge, in which case, the last step of taking the rectangular hull is skipped. This is to ensure that after finite number of refinement (essentially, when the abstract locations and edges are singleton sets), the abstract automaton is identical to the concrete automaton.

Refinement. The refinement algorithm essentially constructs a new control abstraction by splitting the equivalence class of locations and edges near the point of infeasibility. We present a specific method to refine along the above lines. Let us define L_2 to be the set of locations appearing in $Reach_{\pi_A,\alpha}(\hat{k}+1)$. If the infeasibility edge corresponds to an abstract edge e, then we also define E and L_1. E is the set of concrete edges in $\alpha^{-1}(e)$, whose target is a location in L_2. And L_1 is the locations appearing in $Reach_{\pi_A,\alpha}(\hat{k})$ which is a source of some edge in E. We then split the equivalence classes such that the elements in L_1, L_2 and E appear in singleton sets. The intuition behind the above construction is that the only elements which effect satisfaction of Equation 1 are those in the above sets. The above splitting will force the refined system to be equivalent to the concrete system locally, and hence Equation 1 is trivially satisfied.

Let us consider the counter-example $\pi_A = l_1 e_2 l_{2,3} e_{3,4} l_4$. The infeasibility index $\hat{k}$ corresponds to the time transition from $l_{2,3}$ to $l_{2,3}$. In more detail, $FReach_{\pi_A}(\hat{k})$ and $FReach_{\pi_A}(\hat{k}+1)$ are the regions C given by the constraints $x \in [0,1], y \in [0,1]$ and D given by the constraints $x \in [0,1], y \in [0,4], x \geq y-3$, respectively. However, $Reach_{\pi_A,\alpha}(\hat{k}+1)$ is given by $l_2 \times \{(1,4)\}$. And taking

the predecessor of $l_2 \times \{(1,4)\}$ with respect to $\dot{x} \in [2,2]$ and $\dot{y} \in [3,3]$, has a non-empty intersection with the region C. Hence, we split the equivalence class $\{l_2, l_3\}$ such that l_2 is in a singleton equivalence class. We will also need to split the edges e_3 and e_4 to obtain a consistent partition. Therefore, the refinement step results in the concrete automaton.

The control abstraction based CEGAR will always terminate, since starting with any partition of locations and edges, it is possible to refine the partitions only finitely many times due to the fact that the set of locations and edges is finite. Hence, we obtain a complete CEGAR algorithm for the class of initialized rectangular automata. Further, it is fair.

Theorem 2. *The control abstraction based CEGAR algorithm is complete for the class of initialized rectangular hybrid automata, and is fair.*

4.3 Flow Abstraction Based CEGAR

We present another complete algorithm for the class of initialized rectangular automata which abstracts the continuous dynamics by dropping certain variables or applying a limited form of linear transformation, namely, scaling on the variables.

Abstraction. A flow abstraction preserves the underlying control flow graph. A specification of a flow abstraction provides a subset of the variables of the concrete automaton and a scaling factor, a natural number, for each of the variables in the subset. The abstraction is constructed by first dropping the variables not in the specified subset, that is, only the constraints corresponding to the variables in the subset are retained in the invariants, guards, resets and differential inclusions. Next, the constraints are scaled according to the scaling factors provided. A scaling factor of $k \in \mathbb{N}$ for a variable x involves, replacing every constant c appearing in the constraints involving x by c/k. Note that this step may result in an automaton with rational end-points. Hence, we take the rectangular hull of the sets obtained. The only exception is when the scaling factor is 1, in which case we do not take the rectangular hulls. This is to guarantee that when all the variables are included with a scaling factor of 1, we obtain the concrete automaton. Scaling factor helps in reducing the granularity for the purpose of analyzing the system.

Let us consider the automaton $\mathcal{H}_1$, and a flow abstraction which keeps the variable x with scaling factor of 1. The resulting abstract system is obtained by removing all the constraints involving y from $\mathcal{H}_1$. Let us consider another abstraction in which we keep both the variables, x with a scaling factor of 1 and y with a scaling factor of 2. The resulting abstract automaton is shown in Figure 4. The flow of y in $\mathcal{H}_2$ is given by $\dot{y} \in [3,3]$, which when scaled down by 2 gives the $\dot{y} \in [1.5, 1.5]$, and then taking the rectangular hull of the set gives $\dot{y} \in [1,2]$. Similarly, the guard on e_3 is transformed from $y \in [3,10]$ to $y \in [1,5]$.

Refinement. The refinement algorithm consists of two steps. Broadly, in the first step we choose a subset of variables, and in the second step assign appropriate scaling factors to the chosen variables. We iterate over all subsets of variable in the increasing order of size and check if Equation 1 holds, when the scaling factor for all the variables is taken to be 1. Then we assign a scaling factor for a variable in the chosen set, to be the g.c.d of the previous scaling factor and all the constants appearing in the constraints involving the variable locally, that is, the constants appearing the invariants and flows of the location, if the infeasibility index corresponds to a time transition, otherwise, one considers the constants appearing in the guards and reset of the edge corresponding to the infeasibility index. These conditions ensure that Equation 1 is satisfied.

Let us consider the abstraction $\mathcal{H}_4$ and the counter-example $l_1e_2l_3e_4l_4$. We observe that the infeasibility index corresponds to transition from l_3 to l_3 with time elapse. Hence, we need to add y. However, since the g.c.d of the constants corresponding to location l_3, namely, the differential inclusion $\dot{y} \in [2, 2]$ and the invariant $y \in [-10, 10]$, is 2, we assign a scaling factor of 2 with y. The resulting refinement is the automaton $\mathcal{H}_5$ shown in Figure 4.

Again, the flow abstraction based CEGAR algorithm always terminates, because there are only finite number of refinements starting from any abstraction. Every refinement entails adding a variable or changing the scaling associated with a variable. Note that the new scaling factor is necessarily lower than or equal to the previous scaling factor, because by definition the new scaling factor is a divisor of the previous scaling factor. Since the number of variable is finite, and the scaling factor associated with a newly introduced variable is finite, we obtain that the CEGAR algorithm terminates in a finite number of iterations. Also, the CEGAR algorithm is fair.

Theorem 3. *The flow abstraction based CEGAR algorithm is complete for the class of initialized rectangular hybrid automata, and is fair.*

4.4 Discussion

We obtain a semi-complete algorithm for the class of rectangular hybrid automata by composing all the three algorithms, and a complete algorithm for the class of initialized rectangular automata by composing the last two algorithms. The compositional CEGAR framework provides a convenient method to describe and implement CEGAR algorithms in a modular fashion.

The hybrid abstraction based CEGAR algorithms have the advantage that the various phases of the CEGAR loop are more efficient. As said before, construction of the abstraction is simpler because one can avoid expensive unbounded *Post* computations with respect to time. However, the validation phase as described requires computing unbounded *Pre*. One can avoid unbounded *Pre* computations, by using a small trick. One can add a new clock (a variable x with $\dot{x} = 1$), which forces taking a discrete transition every τ time units. This is achieved by adding self loops on the locations with a guard $x \in [\tau, \tau]$ and reset $x :\in [0, 0]$ and adding the constraint $x \in [0, \tau]$ to the invariant. The new system

is equivalent to the old system in terms of checking safety. However, validating a counter-example of the new system requires computing *Pre* with respect to the time interval $[0, \tau]$.

5 Implementation and Experimental Results

The tool, which we call **H**ybrid **A**bstraction **R**efinement **E**ngine (HARE), implements the CEGAR algorithm in C++. HARE input consists of a hybrid automaton and a single initial abstraction function (which combines all the three different types of abstractions). This function defines the initial abstract hybrid automaton. The default initial abstract automaton has no variables and has three locations—an initial location, an unsafe location, and a third location corresponding to all the other locations of the concrete automaton. This abstract automaton is automatically translated to the input language for HyTech [18] and then model-checked. If HyTech does not produce a counterexample for the safety property, HARE returns the current abstraction. Otherwise, the counterexample is parsed and validated. In order to validate the counter-example, we need to compute *Pre* and *Post* with respect to certain edges and/or time. However, HyTech allows taking *Pre* and *Post* only with respect to the set of all the transitions. Thus, our implementation of the validation involves construction of new hybrid automata corresponding to the counter-example, and calls HyTech's *Pre* and *Post* functions on these automata. These calls to HyTech, at least in part, contribute to the relatively large time that HARE spends in the validation step for some of the case studies. Our implementation consists of a single CEGAR algorithm which is a combination of the the three CEGAR schemes presented in Section 4. It does not correspond to any particular composition of the algorithms, and our presentation of the algorithm as a composition is purely for highlighting the ideas in the implementation in a readable manner.

5.1 Experimental Results

Our experimental evaluation of HARE (see Table 1) is based on five classes of examples:

Class 1: BILL_n models a ball in an n-dimensional bounded reflective rectangle. The unsafe set is a particular point in the bounded rectangle.
Class 2: NAV_n models the motion of a point robot in an $n \times n$ grid where each region in the grid is associated with a rectangular vector field. When the robot is in a region, its motion is described by the flow equations of that region. The unsafe set is a particular set of regions. NAV_n_A and NAV_n_B represent the two different configurations of the vector fields, the initial and the unsafe regions. NAV_n_C models two robots on the same the $n \times n$ grid with different initial conditions; the unsafe set being the set of states where the two robots simultaneously reach the same unsafe region.
Class 3: SATS_n models a distributed air traffic control protocol with n aircraft presented in [21]. The model of each aircraft captures several (8 or 10) physical

regions in the airspace where the aircraft can be located, such as the left holding region at 3K feet, the left approach region, the right missed-approach region, the runway, etc. The continuous evolution of the aircraft are described by rectangular dynamics within each region. An aircraft transitions from one to another region based on the rules defined by the traffic control protocol, which involves the state of the current aircraft and also other aircrafts. Thus, the automata for the different aircraft communicate through shared variables. The safety property requires that the distance between two aircraft is never less than a safety constant c. We have worked on two variants of SATS: SATS_n_S models just one side of the airspace and the full SATS_n_C has two sides.
Class 4: FISME_n models Fischer's timing-based mutual exclusion algorithm with n concurrent processes.
Class 5: ZENO is a variant of the well-known 2D bouncing ball system where the system has zeno executions.

Table 1. The columns (from left) show the problem name, sizes of the concrete and final abstract hybrid automaton, number of CEGAR iterations, time taken for validation, time taken for refinement, total time by HARE and the time taken by HyTech

Problem	Conc. size (locs, vars)	Abst. size (locs, vars)	Iter.	Validation (sec)	Abstraction Refinement(sec)	HARE (sec)	HyTech (sec)
BILL_2_A	(6,2)	(4, 1)	1	0.02	0.04	0.06	0.03
BILL_3_A	(8,3)	(4, 1)	1	0.04	0.06	0.1	0.04
NAV_10_A	(100,2)	(6, 2)	4	0.64	0.16	0.8	0.16
NAV_15_A	(225,2)	(6, 2)	4	1.07	0.18	1.25	0.27
NAV_10_B	(100,2)	(5, 1)	4	0.67	0.16	0.83	0.24
NAV_15_B	(225,2)	(5, 1)	4	1.84	0.29	2.13	0.52
NAV_8_C	$(64^2,4)$	$(7^2, 4)$	5	1.45	1.39	2.84	23.54
NAV_10_C	$(100^2,4)$	$(7^2, 4)$	5	2.41	1.51	3.92	58.24
NAV_14_C	$(196^2,4)$	$(7^2, 4)$	5	5.38	1.74	7.12	346.83
SATS_3_S	(512,3)	(320, 2)	4	0.48	1.92	2.40	2.64
SATS_4_S	(4096,4)	(1600, 2)	4	5.25	15.38	20.63	23.75
SATS_5_S	(32786,5)	(8000,2)	4	45.79	106.58	154.17	189.65
SATS_3_C	(1000,4)	(500, 2)	5	2.04	3.82	5.86	6.26
SATS_4_C	(10000,5)	(2500, 2)	5	22.25	41.37	63.98	76.63
FISME_2	$(4^2,4)$	(9, 4)	4	0.03	0.07	0.1	0.02
FISME_3	$(4^3,5)$	(36, 4)	4	0.44	1.34	1.78	1.98
FISME_4	$(4^4,6)$	(144, 4)	4	28.27	22.21	50.48	78.23
ZENO_BOX	(7,2)	(5,1)	1	0.04	0.04	0.08	—

It is clear from Table 1 that HARE produces relatively small abstractions: in some cases with two orders of magnitude reduction in the number of locations, and often reducing the continuous state space by one or two dimensions. In the extreme case of NAV_n_A, an abstraction with 6 discrete states is found in 4 iterations, independent of the size of the grid. This is not too surprising in

hindsight because the final abstraction clearly illustrates why only a constant number of control locations can reach the unsafe region in this example, and it successfully lumps all the unreachable locations together. Yet, the total verification time is better for HyTech for NAV_n_A and NAV_n_B than HARE primarily because, as discussed earlier, HARE makes numerous calls to HyTech not only for model checking the abstract automaton but also for the validation and the refinement refinement steps. Note that the time taken for abstraction refinement is comparable to that of the time taken for direct verification by HyTech. HARE 's advantage is apparent in the case of NAV_C_*, SATS, and FISME, where the system consists of several automata evolving in parallel. In NAV_C, for example, since the motion of each of the robots can be abstracted into a simpler automaton with less number of discrete locations, the state space of the composition of these abstract automaton is reduced dramatically (exponentially in the number of robots) and this is apparent in the differences in the running time.

In SATS, the time taken for validation is less compared to the time taken for abstraction and refinement steps. This is primarily because of the nature of the system. In SATS case study, the time taken to verify the abstractions is considerable while compared to other case studies.

The advantage of variable-hiding abstraction is apparent in ZENO (HyTech does not terminate in this case), as a subset of variables are sufficient to infer the safety of the system. We believe that in a complex hybrid automaton, with several components, adding the sufficient number of variables and abstracting the state space of hybrid automaton will yield better abstractions. All of this suggests, a direction of research, one we plan on pursuing, where the model-checker is more closely integrated with an abstraction refinement tool such as HARE.

Acknowledgements. The authors would like to thank Nima Roohi for comments on the draft. This work was supported in part by NSF Grant CNS 1016791.

References

1. HARE, https://wiki.cites.uiuc.edu/wiki/display/MitraResearch/HARE
2. Alur, R., Dang, T., Ivančić, F.: Counter-Example Guided Predicate Abstraction of Hybrid Systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 208–223. Springer, Heidelberg (2003)
3. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid system (2002)
4. Ball, T., Rajamani, S.: 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)
5. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. JFCS 14(4), 583–604 (2003)
6. Clarke, E.M., 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)

7. 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)
8. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: ICSE, pp. 439–448 (2000)
9. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic Abstraction Refinement for Timed Automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007)
10. Doyen, L., Henzinger, T.A., Raskin, J.-F.: Automatic Rectangular Refinement of Affine Hybrid Systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 144–161. Springer, Heidelberg (2005)
11. Fehnker, A., Clarke, E.M., Jha, S., Krogh, B.: Refining Abstractions of Hybrid Systems Using Counterexample Fragments. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 242–257. Springer, Heidelberg (2005)
12. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
13. 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)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: POPL 2002, pp. 58–70 (2002)
15. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Proc. of STOC, pp. 373–382 (1995)
16. Henzinger, T.A.: The theory of hybrid automata. In: LICS, pp. 278–292 (1996)
17. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: A Model Checker for Hybrid Systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–483. Springer, Heidelberg (1997)
18. Henzinger, T.A., Ho, P.-H., Howard, W.-T.: Hytech: A Model Checker for Hybrid Systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–483. Springer, Heidelberg (1997)
19. Holzmann, G., Smith, M.: Automating software feature verification. Bell Labs Technical Journal 5(2), 72–87 (2000)
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. Munoz, C.A., Dowek, G., Carreo, V.: Modeling and verification of an air traffic concept of operations. In: ISSTA, pp. 175–182 (2004)
22. Prabhakar, P., Duggirala, S., Mitra, S., Viswanathan, M.: Hybrid automata-based cegar for rectangular hybrid automata, http://software.imdea.org/people/pavithra.prabhakar/Papers/vmcai2013tr.pdf
23. Segelken, M.: Abstraction and Counterexample-Guided Construction of ω-Automata for Model Checking of Step-Discrete Linear Hybrid Models. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007)
24. Sorea, M.: Lazy Approximation for Dense Real-Time Systems. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRFTS 2004. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004)

Quantifying Information Leakage of Randomized Protocols*

Fabrizio Biondi[1], Axel Legay[2], Pasquale Malacaria[3], and Andrzej Wąsowski[1]

[1] IT University of Copenhagen, Denmark
[2] INRIA Rennes, France
[3] Queen Mary University of London, United Kingdom

Abstract. The quantification of information leakage provides a quantitative evaluation of the security of a system. We propose the usage of Markovian processes to model and analyze the information leakage of deterministic and probabilistic systems. We show that this method generalizes the lattice of information approach and is a natural framework for modeling refined attackers capable to observe the internal behavior of the system. We also use our method to obtain an algorithm for the computation of channel capacity from our Markovian models. Finally, we show how to use the method to analyze timed and non-timed attacks on the Onion Routing protocol.

1 Introduction

Quantification of information leakage is a recent technique in security analysis that evaluates the amount of information about a secret (for instance about a password) that can be inferred by observing a system. It has sound theoretical bases in Information Theory [1,2]. It has also been successfully applied to pratical problems like proving that patches to the Linux kernel effectively correct the security errors they address [3]. It has been used for analysis of anonymity protocols [4,5] and analysis of timing channels [6,7]. Intuitively, *leakage of confidential information of a program is defined as the difference between the attacker's uncertainty about the secret before and after available observations about the program* [1].

The underlying algebraic structure used in leakage quantification for *deterministic* programs is the *lattice of information* (LoI) [1]. In the LoI approach an attacker is modelled in terms of possible observations of the system she can make. LoI uses an equivalence relation to model how precisely the attacker can distinguish the observations of the system. An execution of a program is modeled as a relation between inputs and observables. In this paper we follow the LoI approach but take a process view of the system. A process view of the system is a more concise representation of behaviour than an observation relation. Moreover a process view does not require that the system is deterministic, which allows us

* The research presented in this paper has been partially supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 68–87, 2013.

to handle randomized protocols—for the first time using a generic, systematic and implementable LoI-based methodology.

We use Markov Decision Processes to represent the probabilistic partial-information semantics of programs, using the nondeterminism of the model for the choices that depend on the unknown secret. We define the leakage directly on such model. With our method we can distinguish the inherent randomness of a randomized algorithm from the unpredictability due to the lack of knowledge about the secret. We exploit this distinction to quantify leakage only for the secret, as the information leakage about the random numbers generated is considered uninteresting (even though it is an information in information theoretical sense). We thus work with both deterministic and randomized programs, unlike the previous LoI approach.

We give a precise encoding of an *attacker* by specifying her prior knowledge and observational capabilities. We need to specify which of the logical states of the system can be observed by the attacker and which ones he is able to distinguish from each other. Given a program and an attacker we can calculate the leakage of the program to the attacker.

We also show how to turn the leakage computation into leakage optimization: we compute the maximum leakage over all possible prior information of attackers *ceteris paribus*, or in other words, the leakage for the worst possible attacker without specifying the attacker explicitly. This maximum leakage is known as the *channel capacity* of the system [8]. Since we are able to model a very large class of attackers the obtained channel capacity is robust. Computing channel capacity using this method requires solving difficult optimization problems (as the objective is nonlinear), but we show how the problem can be reduced to standard reward optimization techniques for Markovian models for a class of interesting examples.

Our method can be applied to finite state systems specified using a simple imperative language with a randomization construct. It can also be used for systems modeled directly as Markov Decision Processes. We demonstrate the technique using an MDP model of the known Onion Routing protocol [9], showing that we can obtain the channel capacity for a given topology from an appropriate Markov Decision Process describing the probabilistic partial information behavior of the system. Also, our behavioral view of the system allows us to encode an attacker with time-tracking capabilities and prove that such an attacker can leak more information than the canonical attacker that only observes the traffic on the compromised nodes. Timing-based attacks to the Onion Routing protocol have been implemented before [10,11], but to our best knowledge the leakage of timing-attacks has not been quantified before.

Our contributions include:

– A method for modeling attack scenarios consisting of process models of systems and observation models of attackers, including a simple partial-observability semantics for imperative programs, so that these models can also be obtained from code.

- A definition of leakage that generalizes the LoI approach to programs with randomized choices (strictly including the class of deterministic programs), and dually the first application of the LoI approach to process specifications of systems.
- A method for computing leakage for scenarios modeled as described above. The method is fully implementable.
- A method to parameterize the leakage analysis on the attacker's prior information about the secret, to allow the computation of channel capacity by maximizing an equation characterizing leakage as a function of prior information.
- The worst-case analysis of the Onion Routing protocol when observed by non time-aware and time-aware attackers able to observe the traffic passing through some compromised nodes.

The paper proceeds as follows. Section 2 provides the core background on probabilistic systems and the LoI approach. Section 3 gives an overview of our new leakage quantification method. The non-obvious steps are further detailed in Sections 4–6. In Sect. 7 we explain how to use the method for computing channel capacity, and we use this technique to analyze leakage in the onion routing protocol against untimed and timing attacks (Sect. 8). We discuss the related work (Sect. 9) and conclude (Sect. 10).

2 Background

2.1 Markovian Models

Definition 1. *A tuple $\mathcal{C} = (S, s_0, P)$ is a Markov Chain (MC), if S is a finite set of states, $s_0 \in S$ is the initial state and P is an $|S| \times |S|$ probability transition matrix, so $\forall s, t \in S.\ P_{s,t} \geq 0$ and $\forall s \in S.\ \sum_{t \in S} P_{s,t} = 1$.*

The probability of transitioning from any state s to a state t in k steps can be found as the entry of index (s, t) in P^k [12]. We call $\pi^{(k)}$ the probability distribution vector over S at time k and $\pi_s^{(k)}$ the probability of visiting the state s at time k; note that $\pi^{(k)} = \pi_0 P^k$, where $\pi_s^{(0)}$ is 1 if $s = s_0$ and 0 otherwise.

A state $s \in S$ is *absorbing* if $P_{s,s} = 1$. In the figures we will not draw the looping transition of the absorbing states, to reduce clutter.

Let $\xi(s, t)$ denote the *expected residence time* in a state t in an execution starting from state s given by $\xi(s, t) = \sum_{n=0}^{\infty} P_{s,t}^n$. We will write ξ_s for $\xi(s_0, s)$.

Given a Markov chain $\mathcal{C} = (S, s_0, P)$ let a *discrimination relation* $\mathcal{R}$ be an equivalence relation over S. Given $\mathcal{C}$ and $\mathcal{R}$ define the *quotient of $\mathcal{C}$ by $\mathcal{R}$* as a new Markov chain $\mathcal{C}/\mathcal{R} = (S/\mathcal{R}, s_0', P')$ where

- $S/\mathcal{R}$ is the set of the equivalence classes of S induced by $\mathcal{R}$
- s_0' is the equivalence class of s_0

– $P' : S/\mathcal{R} \times S/\mathcal{R} \to [0,1]$ is a probability transition function between equivalence classes of $S/\mathcal{R}$ such that

$$\forall c, d \in S/\mathcal{R}.\ P'_{c,d} = \frac{1}{|c|} \sum_{\substack{s \in c \\ t \in d}} P_{s,t}$$

Given k Markov chains $\mathcal{C}^1 = (S^1, s_0^1, P^1)$,...,$\mathcal{C}^k = (S^k, s_0^k, P^k)$ their synchronous *parallel composition* is a MC $\mathcal{C} = (S, s_0, P)$ where S is $S^1 \times ... \times S^k$, s_0 is $s_0^1 \times ... \times s_0^k$ and $P_{s^1 \times ... \times s^k, t^1 \times ... \times t^k} = \prod_{i=1}^{k} P_{s^i,t^i}$.

Definition 2. *A* Markov Decision Process *(MDP) is a tuple* $\mathcal{P} = (S, s_0, P, \Lambda)$ *where* S *is a finite set of states containing the initial state* s_0*,* Λ_s *is the finite set of available actions in a state* $s \in S$ *and* $\Lambda = \bigcup_{s \in S} \Lambda_s$*, and* $P : S \times \Lambda \times S \to [0,1]$ *is a transition probability function such that* $\forall s, t \in S. \forall a \in \Lambda_s.\ P(s,a,t) \geq 0$ *and* $\forall s \in S. \forall a \in \Lambda_s.\ \sum_{t \in S} P(s,a,t) = 1$.

We we will write $s \xrightarrow{a} [P_1 \mapsto t_1, ..., P_n \mapsto t_n]$ to denote that in state $s \in S$ the system can take an action $a \in \Lambda_s$ and transition to the states $t_1, ..., t_n$ with probabilities $P_1, ..., P_n$.

We will enrich our Markovian models with a finite set $\mathtt{V}$ of integer-valued variables, and an assignment function $A : S \to \mathbb{Z}^{|\mathtt{V}|}$ assigning to each state the values of the variables in that state. We will use the expression $\mathtt{v}_s$ to denote the value of the variable $\mathtt{v} \in \mathtt{V}$ in the state $s \in S$. Later we will use the values of the variables to define the discrimination relations, as explained in Section 6.

2.2 Reward and Entropy of a Markov Chain

A real-valued reward functions on the transitions of a MC $\mathcal{C} = (S, s_0, P)$ is a function $R : S \times S \to \mathbb{R}$. Given a reward function on transitions, the expected reward $R(s)$ for a state $s \in S$ can be computed as $R(s) = \sum_{t \in S} P_{s,t} R(s,t)$, and the expected total reward $R(\mathcal{C})$ of $\mathcal{C}$ as $R(\mathcal{C}) = \sum_{s \in S} R(s) \xi_s$.

The entropy of a probability distribution is a measure of the unpredictability of the events considered in the distribution [13]. Entropy of a discrete distribution over the events $x \in X$ is computed as $\sum_{x \in X} \mathbf{P}(x) \log_2 \frac{1}{\mathbf{P}(x)} = -\sum_{x \in X} \mathbf{P}(x) \log_2 \mathbf{P}(x)$. We will sometimes write $H(\mathbf{P}(x_1), \mathbf{P}(x_2), .., \mathbf{P}(x_n))$ for the entropy of the probability distribution over $x_1, ..., x_n$.

Since every state s in a MC $\mathcal{C}$ has a discrete probability distribution over the successor states we can calculate the entropy of this distribution. We will call it *local entropy*, $L(s)$, of s: $L(s) = -\sum_{t \in S} P_{s,t} \log_2 P_{s,t}$. Note that $L(s) \leq \log_2(|S|)$.

As a MC $\mathcal{C}$ can be seen as a discrete probability distribution over all of its possible traces, we can assign a single entropy value $H(\mathcal{C})$ to it. The global entropy $H(\mathcal{C})$ of $\mathcal{C}$ can be computed by considering the local entropy $L(s)$ as the expected reward of a state s and then computing the expected total reward of the chain [14]:

$$H(\mathcal{C}) = \sum_{s \in S} L(s) \xi_s$$

2.3 Lattice of Information

Let Σ be a finite set of observables over a deterministic program $\mathcal{P}$. Consider all possible equivalence relations over Σ; each of them represents the discriminating power of an attacker. Given two equivalence relations $\approx, \sim$ over Σ define a refinement ordering as

$$\approx \sqsubseteq \sim \quad \text{iff} \quad \forall \sigma_1, \sigma_2 \in \Sigma \ (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_1 \approx \sigma_2) \tag{1}$$

The ordering forms a *complete lattice* over the set of all possible equivalence relations over Σ [15]: the Lattice of Information (abbreviated as `LoI`).

If $\approx \sqsubseteq \sim$ then classes in $\sim$ refine (split) classes in $\approx$, thus $\sim$ represents an attacker that can distinguish more while $\approx$ represents an attacker that can distinguish less observables.

By equipping the set Σ with a probability distribution we can see an equivalence relation as a random variable (technically it is the set theoretical kernel of a random variable but for information theoretical purposes can be considered a random variable [1]). Hence the `LoI` can be seen as a lattice of random variables.

The connection between `LoI` and leakage can be illustrated by this simple example: consider a password checking program checking whether the user input is equal to the secret `h`. Then an attacker observing the outcome of the password check will know whether the secret is `h` or not, hence we can model the leakage of such a program with the equivalence relation $\{\{\mathtt{h}\}, \{x | x \neq \mathtt{h}\}\}$.

More generally, observations over a deterministic program $\mathcal{P}$ form an equivalence relation over the possible states of $\mathcal{P}$. A particular equivalence class will be called an observable. Hence an observable is a set of states indistinguishable by an attacker making that observation. If we consider an attacker able to observe the outputs of a program then the random variable associated to a program $\mathcal{P}$ is given by the equivalence relation on any two states σ, σ' from the universe of program states Σ defined by

$$\sigma \simeq \sigma' \iff [\![\mathcal{P}]\!](\sigma) = [\![\mathcal{P}]\!](\sigma') \tag{2}$$

where $[\![\mathcal{P}]\!]$ represents the denotational semantics of $\mathcal{P}$ [16]. Hence the equivalence relation amounts to "having the same observable output". This equivalence relation is nothing else than the set-theoretical kernel of the denotational semantic of $\mathcal{P}$ [17].

Given a random variable associated to an attacker's observations of a deterministic program $\mathcal{P}$ the *leakage* of P is then defined as the Shannon entropy of that random variable. It is easy to show that for deterministic programs such entropy is equal to the difference between the attacker's a priori and a posteriori uncertainty about the secret and that it is zero if and only if the program is secure (i.e. non interferent) [1].

More intentional attackers in the LoI setting are studied in [18,7], however this is the first work where LoI is used to define leakage in a probabilistic setting.

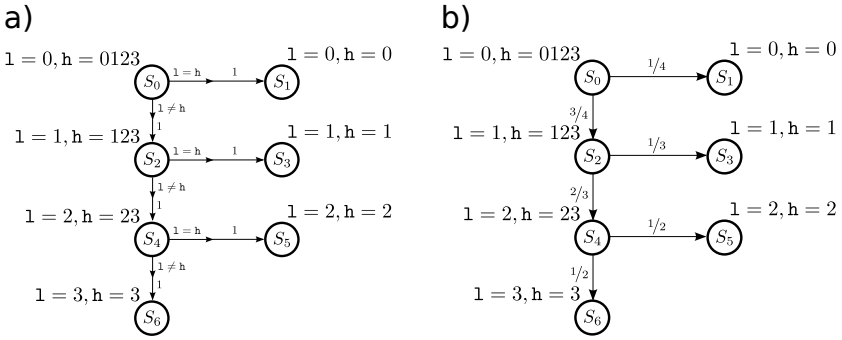


Fig. 1. Simple loop example a) MDP semantics b) MC model

3 Information Leakage of Markov Chains

We begin with an overview of the proposed technique for leakage quantification. It proceeds in five steps, that are all fully automatable for finite state programs. Let a *scenario* be a pair $(\mathcal{P}, \mathcal{A})$, where $\mathcal{P}$ is the system we want to analyze and $\mathcal{A}$ is an attacker. We will call $\mathcal{P}$ the *program*, even if it can be any system suitably modeled as an MDP as explained in Sect. 4.

Step 1: Define a MDP representing $\mathcal{P}$ (Sections 4, 8). We first give a probabilistic semantics to the program in the form of an MDP, in which probabilistic choices are represented by successor state distributions and branching is represented by decision states. This is more or less standard definition of operational semantics for randomized imperative programs.

Example [17]. A program has two variables `l` and `h`. Variable `h` is 2-bit long and private, while variable `l` is public. The attacker can read `l` but not `h`:

```
l = 0; while (l != h) do l = l + 1;
```

The MDP representing the probabilistic partial information semantics of the program is depicted in Fig. 1a. The states in which the system stops and produces an output are encoded with the absorbing states of the MDP, i.e. the states with a probability of transitioning to themselves equal to 1. In the MDP in Fig. 1a states S_1, S_3, S_5 and S_6 are absorbing states.

Step 2: Define the attacker $\mathcal{A}$. An attacker is an external agent observing the system to infer information about its private data. We assume that the attacker knows the implementation of the system (white-box), but is not necessarily able to observe and discriminate all the logical states of the system at runtime. We specify the prior information about the system that the attacker might have, and which system states she can observe and discriminate at runtime.

Definition 3. *An attacker is a triple $\mathcal{A} = (\mathcal{I}, \mathcal{R}_\mathcal{A}, \mathcal{T}_\mathcal{A})$ where $\mathcal{I}$ is a probability distribution over the possible values of the secret encoding the attacker's prior information about it, $\mathcal{R}_\mathcal{A}$ is a discrimination relation over the states of the system*

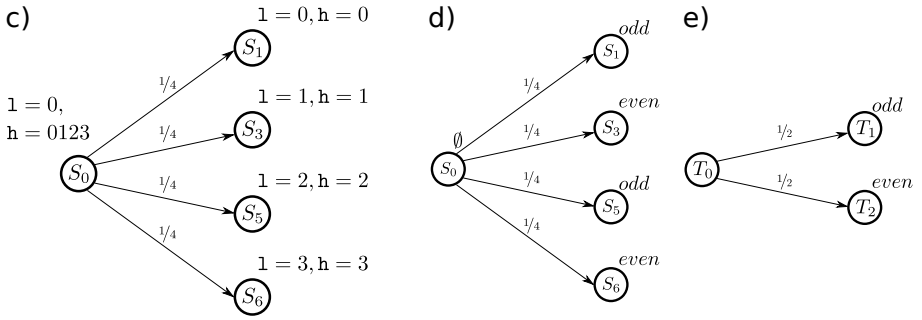


Fig. 2. Simple loop example c) Observable reduction d) Relabeling e) Quotient

in which two states are in the same class iff the attacker cannot discriminate them, and $\mathcal{T}_{\mathcal{A}} \subseteq S$ *is the set of states hidden to the attacker.*

Example. In our example we will use the following attacker: $\mathcal{I} = (1/4, 1/4, 1/4, 1/4)$ (no prior information), $\mathcal{T}_{\mathcal{A}} = (S_2, S_4)$ (cannot observe internal states) and $\mathcal{R}_{\mathcal{A}} = \{(S_1, S_5), (S_3, S_6)\}$ (cannot distinguish states S_1 from S_5 and S_3 from S_6).

Step 3: Resolve the nondeterminism in the MDP. To transform the MDP in a MC, an thus compute leakage, we need to exploit the prior information $\mathcal{I}$ of the attacker. We use it to compute a probability distribution over possible values of private variables in each states of the MDP. To do this for a given state s we just need to normalize $\mathcal{I}$ on the allowed values of the private variables for the state. The probability of the each action $a \in \Lambda_s$ is computed as the probability of the event labelling a given the probability distribution over the values of the secret in s. We will denote the obtained MC by $\mathcal{C}$.

Example. In state S_0 the probability distribution over `h` is $\mathcal{I} = (1/4, 1/4, 1/4, 1/4)$ and `l=0`. The program transitions to state S_1 if `h=l` and to state S_2 if `h`$\neq$`l`. We have that P_{S_0,S_1} is $\mathbf{P}(\mathtt{h} = \mathtt{l}|S_0) = 1/4$ and the probability distribution on `h` in S_1 is $(1, 0, 0, 0)$. Complementarily, P_{S_0,S_2} is $3/4$ and the probability distribution on `h` in S_2 is $(0, 1/3, 1/3, 1/3)$. Figure1b shows the outcome after repeating this step in all states of the MDP of Fig.1a.

Step 4: Hide non-observable states (Sect.5). In the above example the attacker cannot observe the internal states of the system. We expressed this by taking $\mathcal{T}_{\mathcal{A}} = (S_2, S_4)$. Since these states are not observable, we remove them from the MC and redistribute the probability of visiting them to their successors. If a hidden state has no or only hidden successors, it will never produce any observable—we call this event *divergence*. In general we assume that the observer can understand if the program diverges, so divergence is one of the possible outputs of the system. We write $\mathbb{C}$ for the MC resulting from hiding in $\mathcal{C}$ the states of $\mathcal{T}_{\mathcal{A}}$. We call $\mathbb{C}$ the *observable reduction* of the scenario.

Example. Figure2c presents the observable reduction for the running example.

Step 5: Compute the leakage (Sect. 6). From the observable reduction $\mathbb{C}$ and the attacker's discrimination relation $\mathcal{R}_{\mathcal{A}}$ we can compute the leakage for the scenario $(\mathcal{P}, \mathcal{A})$. The definition of leakage for this model is based on the quotient operator for Markov chains. A quotiented MC $\mathcal{C}/\mathcal{R}$ captures the view of the chain when observed by an agent able to distinguish equivalence classes of $\mathcal{R}$. Let $\mathcal{R}_h$ be a discrimination relation that relates states with the same possible values of the secret that is finer than probabilistic bisimulation. Then leakage is the mutual information between the attacker and the system as seen by an agent able to discriminate only states with different values of the secret:

Definition 4. *Let $(\mathcal{P}, \mathcal{A})$ be a scenario, $\mathcal{A} = (\mathcal{I}, \mathcal{R}_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$ an attacker, $\mathbb{C}$ the observable reduction of the scenario and $\mathcal{R}_h = \{(s,t) \in S | h_s = h_t\}$. Then the information leakage of $\mathcal{P}$ to $\mathcal{A}$ is*

$$I(\mathbb{C}/\mathcal{R}_h; \mathbb{C}/\mathcal{R}_{\mathcal{A}}) = H(\mathbb{C}/\mathcal{R}_h) + H(\mathbb{C}/\mathcal{R}_{\mathcal{A}}) - H(\mathbb{C}/\mathcal{R}_{\mathcal{A}} \cap \mathcal{R}_h).$$

Corollary 1. *If $\mathcal{P}$ is a deterministic program, then the leakage is $H(\mathbb{C}/\mathcal{R}_{\mathcal{A}})$.*

Example. Recall that in the running example the attacker is only able to read the parity of `l`. We have that $\mathcal{R}_{\mathcal{A}} = \{(S_1, S_5), (S_3, S_6)\}$. We name the equivalence classes *even* and *odd* and relabel the state with the classes (see Fig. 2d). The quotient $\mathbb{C}/\mathcal{R}_{\mathcal{A}}$ is depicted in Fig. 2e. The program is deterministic, so by Corollary 1 the leakage of the scenario is equivalent to the entropy of such quotient, or 1 bit [14].

4 Handling Randomized Imperative Programs

We give a simple probabilistic partial-observation semantics for an imperative language with randomization. This semantics, akin to abstract interpretation, derives Markovian models of finite state programs automatically. Let all variables be integers of predetermined size and class (public, private) declared before execution. Private variables are read-only, and cannot be observed externally. Denote by l (resp. h) names of public (resp. private) variables; by p reals from $[0; 1]$; by `label` all program points; by `f` (`g`) pure arithmetic (Boolean) expressions. Assume a standard set of expressions and the following statements:

$$\begin{aligned} \textit{stmt} ::= \;& l \texttt{ := f(}l\ldots\texttt{)} \mid l \texttt{ := rand } p \mid \texttt{skip} \mid \texttt{goto label} \mid \\ & \texttt{return} \mid \texttt{if g(}l\ldots\texttt{,}h\ldots\texttt{) then } \textit{stmt-list} \texttt{ else } \textit{stmt-list} \end{aligned}$$

The first statement assigns to a public variable the value of expression `f` depending on other public variables. The second assigns zero with probability p, and one with probability $1-p$, to a public variable. The `return` statement outputs values of all public variables and terminates. A conditional branch first evaluates an expression `g` dependent on private and public variables; the first list of statements is executed if the condition holds, and the second otherwise. For simplicity, all statement lists must end with an explicit jump, as in: `if g(l,h) then ...; goto done; else ...; goto done; done:` ... . Each program can be

$$\frac{pc\text{: } \texttt{skip}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc+1, L, H)]} \qquad \frac{pc\text{: } \texttt{v} := \texttt{f(l)}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc+1, L[^{\texttt{f(l)}}/_{\texttt{v}}], H)]}$$

$$\frac{pc\text{: } \texttt{v := rand } p}{(pc, L, H) \xrightarrow{\top} [p \mapsto (pc+1, L[^{0}/_{\texttt{v}}], H), (1-p) \mapsto (pc+1, L[^{1}/_{\texttt{v}}], H)]}$$

$$\frac{pc\text{: } \texttt{goto label}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (\texttt{label}, L, H)]} \qquad \frac{pc\text{: } \texttt{return}}{(pc, L, H) \xrightarrow{\top} [1 \mapsto (pc, L, H)]}$$

$$\frac{pc\text{: } \texttt{if g(l,h) then la: A else lb: B}}{(pc, L, H) \xrightarrow{\texttt{g(l,h)}} [1 \mapsto (\texttt{la}, L, H|\texttt{g(l,h)})]}$$

$$\frac{pc\text{: } \texttt{if g(l,h) then la: A else lb: B}}{(pc, L, H) \xrightarrow{\neg\texttt{g(l,h)}} [1 \mapsto (\texttt{lb}, L, H|\neg\texttt{g(l,h)})]}$$

Fig. 3. Execution rules in probabilistic partial information semantics

easily transformed to this form. Loops can be added in a standard way as a syntactic sugar.

The probabilistic partial-information semantics assumes an external view of the program, so private variables are not visible. A state in this view is a triple (pc, L, H), where pc is the current program counter, L maps public variables to integer values of the appropriate size, and H maps private variables to sets of their possible values. If the observer knows nothing about a private variable h, the set $H(h)$ holds all the values of h's type. If the observer holds some prior information, or learns through interaction with the system, this set is smaller.

The semantics (Fig. 3) is a small-step operational semantics with transitions from states to distributions over states, labeled by expressions dependent on h (only used for the conditional statement). It generates an MDP over the reachable state space. In Fig. 3, `v`, `l` are public variables and `h` is a private variable. Expressions in rule consequences stand for values obtain in a standard way. $L[^{X}/_{l}]$ denotes substition of X as the new value for l in mapping L. Finally, $H|\texttt{g}$ denotes a restriction of each set of possible values in a mapping H, to contain only values that are consistent with Boolean expression `g`. Observe that the `return` rule produces an absorbing state—this is how we model termination in an MDP. The `rand` rules produces a proper distribution, unlike the other Dirac distributions. The `if` rule produces a nondeterministic decision state.

In the obtained MDP states are labelled by values of public variables and sets of values of private variables. Actions from each state represent the secret-dependent events for the state. Our leakage quantification technique works for any MDP of this shape, even the ones not necessarily obtained from code. In Sect. 8 we will create such a model directly from a topology of the Onion Routing protocol.

1. Take $\mathcal{C} \setminus \mathcal{T} = (S, s_0, \mathbb{P})$ and $\mathbb{P} = P$
2. Add to the MC the divergence state $\uparrow$ with $\mathbb{P}_{\uparrow,\uparrow} = 1$
3. Choose a hidden state $t \in \mathcal{T}$, or terminate if $\mathcal{T}$ is empty
4. Let $Pred(t) = \{s \in S \setminus \{t\} \mid \mathbb{P}_{s,t} > 0\}$ be the set of predecessors of t
5. Let $Succ(t) = \{u \in S \setminus \{t\} \mid \mathbb{P}_{t,u} > 0\}$ be the set of successors of t
6. If $\mathbb{P}_{t,t} = 1$:
   (a) For each state $s \in Pred(t)$ set $\mathbb{P}_{s,\uparrow} = \mathbb{P}_{s,t}$
   (b) Remove t from S and $\mathcal{T}$ and go back to step 3
7. Else
   (a) For each $u \in Succ(t)$ set $\mathbb{P}_{t,u} := \frac{\mathbb{P}_{t,u}}{1-\mathbb{P}_{t,t}}$
   (b) Set $\mathbb{P}_{t,t} = 0$
   (c) For each $s \in Pred(t)$ and $u \in Succ(t)$ set $\mathbb{P}_{s,u} := \mathbb{P}_{s,u} + \mathbb{P}_{s,t}\mathbb{P}_{t,u}$
   (d) Remove t from S and $\mathcal{T}$ and go back to step 3

Fig. 4. Computing $\mathcal{C} \setminus \mathcal{T} = (S \setminus \mathcal{T}, s_0, \mathbb{P})$ for a MC $\mathcal{C} = (S, s_0, P)$ and hidden states $\mathcal{T} \subset S$

5 Hiding Non-observable States

In the simple loop example of Sect. 3 the attacker is unable to observe states S_2 and S_4; we call these non-observable states *hidden*. His view of the system is thus adequately represented by the MC in Fig. 2c. In this figure the probability of transferring from the state S_0 to state S_5 is the probability of reaching S_5 from S_0 in the MC of Fig. 1b *eventually*, so after visiting zero or more hidden states.

Note that the initial state cannot be hidden, as we assume the attacker knows that the system is running. This assumption does not restrict the power of the approach, since one can always model a system, whose running state is unknown, by prefixing its initial state by a pre-start state, making it initial, and hiding the original initial state.

We present the hiding algorithm in Fig. 4. We will overload the symbol $\setminus$ to use for the hiding operation: we write $\mathcal{C} \setminus \mathcal{T}$ for the observable MC obtained from $\mathcal{C}$ by hiding the states in set $\mathcal{T}$. If a system stays in a hidden state forever, we say it *diverges*. Divergence will be symbolized by a dedicated absorbing state named $\uparrow$. Otherwise, we compute the new successor probabilities for t; we accomplish this by setting the probability of transitioning from t to itself to 0 and normalizing the other probabilities accordingly. Then we compute the probability that each of its predecessors s would transition to each of its successors u via t and add it to the transition probability from s to u, and finally we remove t from the MC.

The difference between states that cannot be discriminated and hidden states is of primary importance. The former assumes that the attacker is aware of the existence of such states, and thus knows when the system is in one of them, but is not able to discriminate them because they share the same observable properties. For instance, if the attacker can only read the system's output he will not be able to discriminate between different states that produce the same output. In contrast the attacker has no way to observe the behavior of the system when it is in an hidden state, not even by indirect methods like keeping track of the

discrete passage of time. For instance, if the attacker can only read the system's output, the states of the system that produce no output will be hidden to him.

6 Collapsing Non-discriminable States

Discrimination relations are equivalence relations that we use to encode the fact that some states cannot be observed separately by the attacker, since they share some observable properties. Different attackers are able to observe different properties of the states, and thus discriminate them differently.

The discrimination relation $\mathcal{R}_{\mathcal{A}}$ represents the attacker's inability to determine when the system is in a particolar state due to the fact that different states have the same observable properties. We define equivalence classes based on $\mathcal{R}_{\mathcal{A}}$, and the attacker knows that the system is in one of these classes but not in which state. This is encoded by relabelling the states of the MC with their equivalence classes in $\mathcal{R}_{\mathcal{A}}$ and then quotienting it by $\mathcal{R}_{\mathcal{A}}$.

We need to impose a restriction to $\mathcal{R}_{\mathcal{A}}$, since not all discrimination relations are suitable for encoding attackers: the attacker is always able to discriminate states if they behave differently in the relabelled model. Let $\mathcal{C}^{\mathcal{R}_{\mathcal{A}}}$ be the MC $\mathcal{C}$ in which the states are labeled with their equivalence class in $S/\mathcal{R}_{\mathcal{A}}$. Then $\mathcal{R}_{\mathcal{A}}$ encodes the discrimination relation of an attacker only if the states with the same label in $\mathcal{C}^{\mathcal{R}_{\mathcal{A}}}$ are probabilistically bisimilar.

As a result of this condition, all traces in $\mathcal{C}/\mathcal{R}_{\mathcal{A}}$ are relabelled projections of traces in $\mathcal{C}$. This is fundamental to prevent the attacker from expecting traces that do not appear in the actual computation. It also allows us to generalize the discrimination relation ordering used in the `LoI` approach [1]. Let $\mathcal{A}_1 = (\mathcal{I}_1, \mathcal{T}_{\mathcal{A}_1}, \mathcal{R}_{\mathcal{A}_1})$ and $\mathcal{A}_2 = (\mathcal{I}_2, \mathcal{T}_{\mathcal{A}_2}, \mathcal{R}_{\mathcal{A}_2})$ be two attackers, and define

$$\mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \quad \text{iff} \quad \mathcal{I}_1 = \mathcal{I}_2 \wedge \mathcal{T}_{\mathcal{A}_1} = \mathcal{T}_{\mathcal{A}_2} \wedge \mathcal{R}_{\mathcal{A}_1} \subseteq \mathcal{R}_{\mathcal{A}_2}$$

Theorem 1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two attackers such that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. Then for any program $\mathcal{P}$, the leakage of the scenario $(\mathcal{P}, \mathcal{A}_1)$ is greater or equal then the leakage of the scenario $(\mathcal{P}, \mathcal{A}_2)$.*

Effectively, the attacker that is able to discriminate more states (a language-like qualitative property) is able to leak more information (an information-theoretical quantitative property). The attacker with the highest leakage can discriminate all states, thus its discrimination relation is the identity; the attacker with the lowest leakage cannot discriminate any state from any other, and thus has leakage 0.

The common definition of leakage of the `LoI` approach [2] assumes that the attacker can observe the different output of a deterministic system. It can be easily encoded in our method. Consider a deterministic program $\mathcal{P}$ with a low-level variable `o` encoding the output of the program. Let the an attacker $\mathcal{A}_{I/O}$ have $\mathcal{R}_{\mathcal{A}_{I/O}} = \{(s,t) \in S \times S \mid \mathtt{o}_s = \mathtt{o}_t\}$ and $\mathcal{T}_{\mathcal{A}_{I/O}}$ being the set of all internal states of the MDP semantics of $\mathcal{P}$. The following proposition states that such attacker is the one considered in [2]:

Theorem 2. *Let $(\mathcal{P}, \mathcal{A}_{I/O})$ be a scenario, $\mathcal{A}_{I/O}$ being the attacker defined above. Then $H(\mathcal{C}/\mathcal{R}_{\mathcal{A}_{\mathcal{I}/\mathcal{O}}}) = Leakage(\mathcal{P})$.*

7 Computing Channel Capacity

The method we presented computes the leakage for a scenario, but it is common in security to ask what is the leakage of a given program in the worst-case scenario, i.e. for the scenario with the highest leakage. We consider the maximum leakage over all the attackers with the same discrimination relation $\mathcal{R}_{\mathbb{A}}$ and hidden states $\mathcal{T}_{\mathbb{A}}$ but different prior information $\mathcal{I}$. We define a class of attackers this way because maximizing over all discrimination relations would just conclude that the attacker able to discriminate all states leaks all the information in the system. The maximum leakage for a class of attackers is known as channel capacity, and it is the upper bound to the leakage of the system to any attacker [8]:

Definition 5. *Let $\mathcal{P}$ be a program and $\mathbb{A}$ the class of all attackers with discrimination relation $\mathcal{R}_{\mathbb{A}}$ and hidden states $\mathcal{T}_{\mathbb{A}}$. Let $\hat{\mathcal{A}} \in \mathbb{A}$ be the attacker maximizing the leakage of the scenario $(\mathcal{P}, \mathcal{A})$ for all $\mathcal{A} \in \mathbb{A}$. Then the channel capacity of $\mathcal{P}$ is the leakage of the scenario $(\mathcal{P}, \hat{\mathcal{A}})$.*

To compute it we procede as follows. We first transform the MDP semantics of $\mathcal{P}$ in a parameterized MC with constraints. Then we define a MC and a reward function from it such that the expected total reward of the MC is equivalent to the leakage of the system. Then we extract an equation with constraints characterizing this reward as a function of the prior information $\mathcal{I}$ of the attacker. Finally, we maximize the equation and obtain the maximum leakage, i.e. the channel capacity. In the next Section we will apply this method to compute the channel capacity of attacks to the Onion Routing protocol.

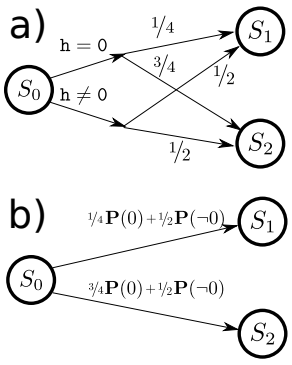


Fig. 5. Reduction from MDP to parameterized MC

Step 1: Find the parameterized MC. We abuse the notation of Markov chain allowing the use of variables in the transition probabilities. This allows us to transform the MDP semantics of a program $\mathcal{P}$ in a MC with the transition probabilities parameterized by the probability of choosing the actions in each state.

Consider the MDP in Fig. 5a; in state S_0 either $\mathtt{h} = \mathtt{0}$ or $\mathtt{h} \neq \mathtt{0}$ and the system moves to the next state with the appropriate transition probability. Let $\mathbf{P}(0)$ and $\mathbf{P}(\neg 0)$ be $\mathbf{P}(\mathtt{h} = \mathtt{0}|S_0)$ and $\mathbf{P}(\mathtt{h} \neq \mathtt{0}|S_0)$ respectively; then we can transform the MDP in the MC in Fig. 5b, with the constraint $\mathbf{P}(0) + \mathbf{P}(\neg 0) = 1$.

We hide the states in $\mathcal{T}_{\mathbb{A}}$ in the MC obtaining the observational reduction $\mathbb{C}$, as described in Sect. 5.

Step 2: Define a reward function for leakage. We want to define a reward function on the parameterized MC such that the expected total reward of the chain is equivalent to the leakage of the system. This step can be skipped if the leakage equation can be obtained directly from the model, like in the examples in the next Section. In the example in Fig. 5 the system is deterministic, so its leakage is equal to its entropy by Corollary 1, and we just need to define the entropy reward function on transitions $R(s,t) = -\log_2 P_{s,t}$, as explained in [14].

For a probabilistic system we need to build another MC by composing $\mathbb{C}/\mathcal{R}_h$, $\mathbb{C}/\mathcal{R}_{\mathbb{A}}$ and $\mathbb{C}/\mathcal{R}_{\mathbb{A}} \cap \mathcal{R}_h$, and we define the leakage reward function on the composed chain:

Theorem 3. *Let $\mathcal{C}$ be the parallel composition of $\mathbb{C}/\mathcal{R}_h$, $\mathbb{C}/\mathcal{R}_{\mathbb{A}}$ and $\mathbb{C}/\mathcal{R}_{\mathbb{A}} \cap \mathcal{R}_h$. Let R be a reward function on the transitions of $\mathcal{C}$ such that*

$$R(s_1 \times s_2 \times s_3, t_1 \times t_2 \times t_3) = \log_2 \frac{P_{s_1,t_1} P_{s_2,t_2}}{P_{s_3,t_3}}.$$

Then the expected total infinite time reward of $\mathcal{C}$ with the reward function R is equivalent to $H(\mathbb{C}/\mathcal{R}_h) + H(\mathbb{C}/\mathcal{R}_{\mathbb{A}}) - H(\mathbb{C}/\mathcal{R}_{\mathbb{A}} \cap \mathcal{R}_h)$ and thus to the leakage.

Step 3: Extract the leakage as an equation. Now that we have a reward function R on the transitions of a MC characterizing the leakage of the system, we need to maximize it. One possible strategy is to extract the explicit equation of the reward of the chain as a function of the transition probabilities, which themselves are a function of the prior information $\mathcal{I}$. For a reward function $R(s,t)$ on transitions the reward for the MC is

$$R(\mathcal{C}) = \sum_{s \in S} R(s)\xi_s = \sum_{s \in S} \left(\sum_{t \in S} P_{s,t} R(s,t) \cdot \sum_{k=0}^{\infty} P_{s_0,s} \right)$$

Since for the leakage reward function $R(s,t)$ is a function of $P_{s,t}$, the transition probabilities are the only variables in the equation.

In the example in Fig. 5 the leakage is equal to the entropy, so the reward function is $R(s,t) = -\log_2 P_{s,t}$ and the leakage equation is

$$\begin{aligned} \mathcal{R}(\mathcal{C}) = &- (\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2) \log ((\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2)) - \\ &- (3\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2) \log ((3\mathbf{P}(0)/4 + \mathbf{P}(\neg 0)/2)) \end{aligned} \quad (3)$$

under the constraint above.

Step 4: Maximize the leakage equation Maximizing the extracted constrained leakage equation computes the channel capacity of the system. This can be done with any maximization method. Note that in general the strategy maximizing this reward function will be probabilistic, and thus will have to be approximated numerically. In the cases in which the maximum leakage strategy is deterministic, an analytical solution can be defined via Bellman equations. This case is more complex that standard reward maximization for MDPs, since the strategy

in every state must depend on the same prior information $\mathcal{I}$, and this is a global constraint that cannot be defined in a MDP. A theoretical framework to automate this operation is being studied, but most cases are simple enough to not need it, like the examples in the next Section.

8 Onion Routing

8.1 Case: Channel Capacity of Onion Routing

Onion Routing [9] is an anonymity protocol designed to protect the identity of the sender of a message in a public network. Each node of the network is a router and is connected to some of the others, in a directed network connection topology; the topology we consider is the depicted in Fig. 6. When one of the nodes in the topology wants to send a message to the receiver node R, it initializes a path through the network to route the message instead of sending it directly to the destination. The node chooses randomly one of the possible paths from itself to R, respecting the following conditions:

1. No node can appear in the path twice.
2. The sender node cannot send the message directly to the receiver.
3. All paths have the same probability of being chosen.

If some nodes are under the control of an attacker, he may try to gain information about the identity of the sender. In this example node 3 is a compromised node; the attacker can observe the packets transitioning through it, meaning that when a message passes through node 3 the attacker learns the previous and next node in the path. The goal of the attacker is to learn the identity of the sender of the message; since there are 4 possible senders, this is a 2-bit secret.

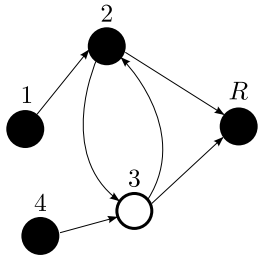


Fig. 6. Network topology for Onion Routing

h	Path	o	P(o|h)
$1(h_1)$	$1 \to 2 \to R$	NN	$\frac{1}{2}$
	$1 \to 2 \to 3 \to R$	$2R$	$\frac{1}{2}$
$2(h_2)$	$2 \to 3 \to R$	$2R$	1
$3(h_3)$	$3 \to 2 \to R$	$N2$	1
$4(h_4)$	$4 \to 3 \to R$	$4R$	$\frac{1}{2}$
	$4 \to 3 \to 2 \to R$	42	$\frac{1}{2}$

Fig. 7. Onion Routing paths, observations and probabilities

Figure 7 summarizes the possible secrets of the protocol, the corresponding paths, the observation for each path assuming node 3 is compromised and the probability that a given sender will choose the path.

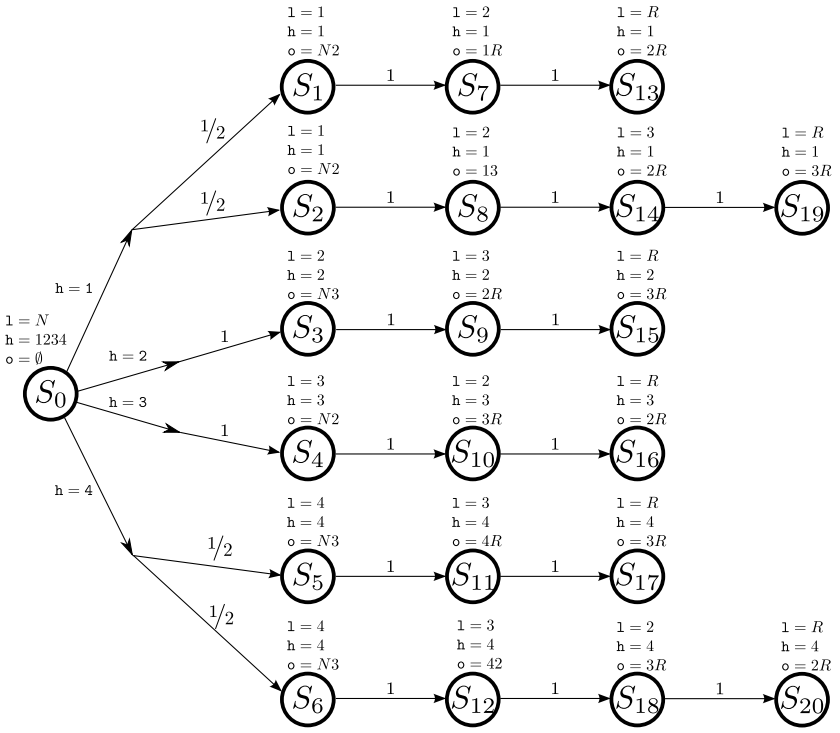


Fig. 8. Markov Decision Process for Onion Routing

We give directly the MDP semantics of the system in Fig.8; its WHILE code is not shown for simplicity. The prior information $\mathcal{I}$ of the attacker consists of the prior probabilities he assigns to the identity of the sender; we use h_i to denote $\mathbf{P}(\mathtt{h} = i)$, for $i = 1...4$. Clearly $h_1 + h_2 + h_3 + h_4 = 1$. The full system is represented in Fig.8, parameterized on the h_i parameters. Each state is labelled with the low-level variables $\mathtt{l}$ and $\mathtt{o}$ and the confidential variable $\mathtt{h}$. Variable $\mathtt{l}$ represents the name of the node being visited in the Onion Routing topology, $\mathtt{o}$ represents the observables in that node (the nodes before and after it in the path), and $\mathtt{h}$ the name of the sender of the message.

Since the attacker can observe only node 3, all states with $\mathtt{l} \neq 3$ except the initial state are unobservable τ-states. We reduce the chain accordingly; the resulting observational reduction is shown in Fig.9a. We call it $\mathbb{C}$. Note that one of the paths does not pass through node 3, so if that path is chosen the attacker will never observe anything; in that case the system diverges. We assume that the attacker can recognize this case, using a timeout or similar means.

To compute the leakage we need also to define $\mathcal{R}_h$ and $\mathcal{R}_\mathcal{A}$. This is straightforward; $\mathcal{R}_h$ is $((s,t) \in (S \times S) | \mathtt{h}_s = \mathtt{h}_t)$ and $\mathcal{R}_\mathcal{A}$ is $((s,t) \in (S \times S) | \mathtt{o}_s = \mathtt{o}_t)$. The resulting MCs $\mathbb{C}/\mathcal{R}_h$ and $\mathbb{C}/\mathcal{R}_\mathcal{A}$ are shown in Fig.9bc. Note that $\mathbb{C}/\mathcal{R}_h \cap \mathcal{R}_\mathcal{A} = \mathbb{C}$.

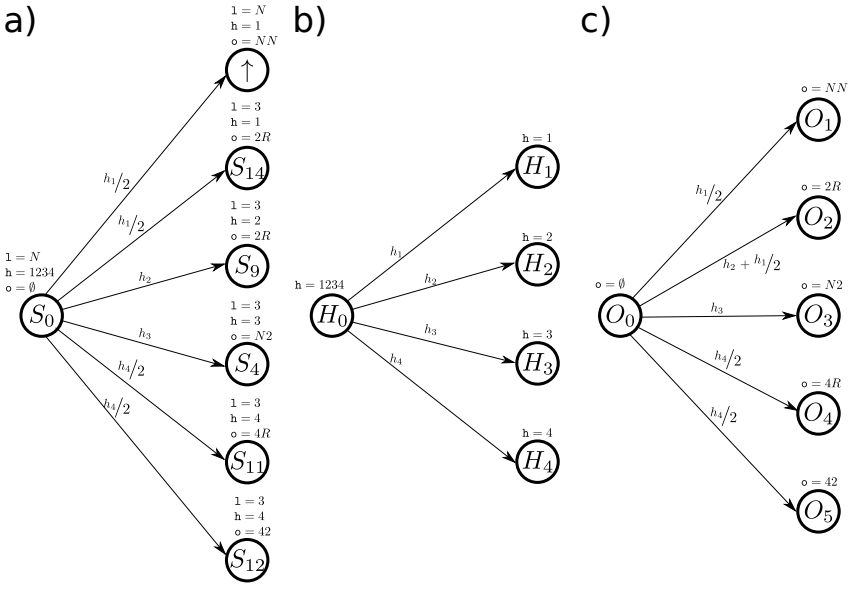


Fig. 9. Markov chains for Onion Routing: a) Observable reduction $\mathbb{C}$ b) $\mathbb{C}/\mathcal{R}_h$ c) $\mathbb{C}/\mathcal{R}_{\mathcal{A}}$

Since the system is very simple, we can extract the leakage equation directly from Def.4. The leakage parameterized on $\mathcal{I}$ is

$$\begin{aligned} &H(\mathbb{C}/\mathcal{R}_h) + H(\mathbb{C}/\mathcal{R}_{\mathcal{A}}) - H(\mathbb{C}/\mathcal{R}_{\mathcal{A}} \cap \mathcal{R}_h) = \\ &= H(h_1, h_2, h_3, h_4) + H(\frac{h_1}{2}, \frac{h_1}{2} + h_2, h_3, \frac{h_4}{2}, \frac{h_4}{2}) - \\ &\quad H(\frac{h_1}{2}, \frac{h_1}{2}, h_2, h_3, \frac{h_4}{2}, \frac{h_4}{2}) \end{aligned} \tag{4}$$

Under constraints $0 \leq h_i \leq 1$ and $h_1 + h_2 + h_3 + h_4 = 1$ it has its maximum of 1.819 bits at $h_1 = 0.2488$, $h_2 = 0.1244$, $h_3 = 0.2834$, $h_4 = 0.2834$, thus these are the channel capacity and the attacker with highest leakage.

8.2 Case: Channel Capacity of Discrete Time Onion Routing

Due to our intensional view of the system, we can naturally extend our analysis to integrate timing leaks. Time-based attacks on the Tor implementation of the Onion Routing network have been proven to be effective, particularly in low-latency networks [10,11]. We show how to quantify leaks for an attacker capable to make some timing observations about the network traffic.

In this example there are two compromised nodes, A and B, and the attacker is able to count how many time units pass between the message being forwarded by A and the message arriving in B. The topology of the network is shown in Fig.10 and the relative paths, observations and probabilities in Fig.11. We will ignore messages departing from the compromised nodes A and B for simplicity.

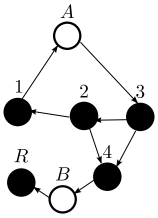


Fig. 10. Network topology for Timed Onion Routing

h	Path	o	P(O|h)
$1(h_1)$	$1 \to A \to 3 \to 4 \to B \to R$	$13, 4R$	$\frac{1}{2}$
	$1 \to A \to 3 \to 2 \to 4 \to B \to R$	$13, 4R$	$\frac{1}{2}$
$2(h_2)$	$2 \to 4 \to B \to R$	$NN, 4R$	$\frac{1}{2}$
	$2 \to 1 \to A \to 3 \to 4 \to B \to R$	$13, 4R$	$\frac{1}{2}$
$3(h_3)$	$3 \to 4 \to B \to R$	$NN, 4R$	$\frac{1}{2}$
	$3 \to 2 \to 4 \to B \to R$	$NN, 4R$	$\frac{1}{2}$
$4(h_4)$	$4 \to B \to R$	$NN, 4R$	1

Fig. 11. Timed Onion Routing paths, observations and probabilities

We add to the system a low-level variable $\mathtt{t}$ that represents the passage of the time between the message passing by A and passing by B. Variable $\mathtt{t}$ is initialized to 0 when the message passes by A and increased by 1 at each subsequent step. We will analyze the difference of leakage between the attacker $\mathcal{A}_\mathcal{T}$ that can discriminate states with different values of $\mathtt{t}$ and the attacker $\mathcal{A}_\mathcal{N}$ that does not have this power.

Both attackers are able to observe nodes A and B, so they have the same hidden states. Their observable reduction $\mathbb{C}$ of the system is the same, depicted in Fig. 12a. The secret's discrimination relation is also the same: $\mathcal{R}_h$ is $((s,t) \in (S \times S) | \mathtt{h}_s = \mathtt{h}_t)$, and the resulting quotient $\mathbb{C}/\mathcal{R}_h$ is depicted in Fig. 12b.

The two attackers have two different discrimination relations. For the attacker $\mathcal{A}_\mathcal{N}$, who is not able to keep count of the discrete passage of time, the relation is $\mathcal{R}_{\mathcal{A}_\mathcal{N}} = ((s,t) \in (S \times S) | \mathtt{o}_s = \mathtt{o}_t)$, while for the time-aware attacker $\mathcal{A}_\mathcal{T}$ it is $\mathcal{R}_{\mathcal{A}_\mathcal{T}} = ((s,t) \in (S \times S) | \mathtt{o}_s = \mathtt{o}_t \wedge \mathtt{t}_s = \mathtt{t}_t)$. The resulting MCs $\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}}$ and $\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}}$ are shown in Fig. 13.

Note that since the time-aware attacker has strictly more discriminating power, since $\mathcal{R}_{\mathcal{A}_\mathcal{T}} \subseteq \mathcal{R}_{\mathcal{A}_\mathcal{N}}$, we expect that he will leak more information. We show now how to validate this intuition by computing the difference of the leakage between $\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{N}$. The difference of the leakage between the two attackers is

$$
\begin{aligned}
& I(\mathbb{C}/\mathcal{R}_h; \mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}}) - I(\mathbb{C}/\mathcal{R}_h; \mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}}) = \\
& H(\mathbb{C}/\mathcal{R}_h) + H(\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}}) - H(\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}} \cap \mathcal{R}_h) - H(\mathbb{C}/\mathcal{R}_h) - \\
& \qquad - H(\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}}) + H(\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}} \cap \mathcal{R}_h) = \\
& H(\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}}) - H(\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}}) = \\
& H\left(h_1 + \frac{h_2}{2}, \frac{h_2}{2} + h_3 + h_4\right) + \left(h_1 + \frac{h_2}{2}\right) H\left(\frac{1}{3}, \frac{2}{3}\right) - \\
& \qquad - H\left(h_1 + \frac{h_2}{2}, \frac{h_2}{2} + h_3 + h_4\right) = \\
& \left(h_1 + \frac{h_2}{2}\right) H\left(\frac{1}{3}, \frac{2}{3}\right) \approx \\
& 0.91829 \left(h_1 + \frac{h_2}{2}\right)
\end{aligned}
\tag{5}
$$

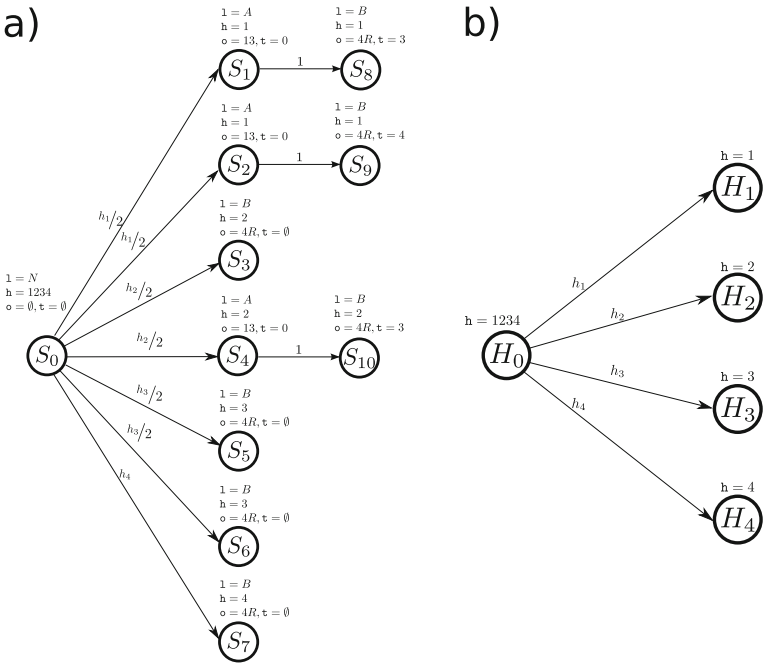


Fig. 12. Markov chains for Timed Onion Routing: a) Observable reduction $\mathbb{C}$ b) $\mathbb{C}/\mathcal{R}_h$

showing that the time-aware attacker $\mathcal{A}_{\mathcal{T}}$ leaks $\approx 0.91829\left(h_1 + \frac{h_2}{2}\right)$ bits of information more than the time-unaware attacker $\mathcal{A}_{\mathcal{N}}$.

9 Related Work

Alvim, Andrés and Palamidessi [19] study leakage and channel capacity of interactive systems where secrets and observables can alternate during the computation.

Chen and Malacaria study leakage and channel capacity of traces and subtraces of programs [18], and, in [20], consider transition systems with particular attention to multi-threaded programs. They use Bellman equations to determine the minimal and maximal leakage. None of these works however deal explicitly with Markov Chains and randomized systems.

Intensional aspects of systems like timing leaks have been investigated by Köpf et al. in [7,6] and more recent work by Köpf, Mauborgne and Ochoa has investigated caching leaks [21].

Channel capacity for the Onion Routing protocol has been first characterized by Chen and Malacaria using Lagrange multipliers [5].

Recently Alvim et al. [22] have proposed a generalization of min-leakage by encapsulating it in problem-dependent gain functions. They suggest a generalization of LOI which would be interesting to compare with our work. On the other hand the use of alternative measure of leakage like g-leakage is a relatively

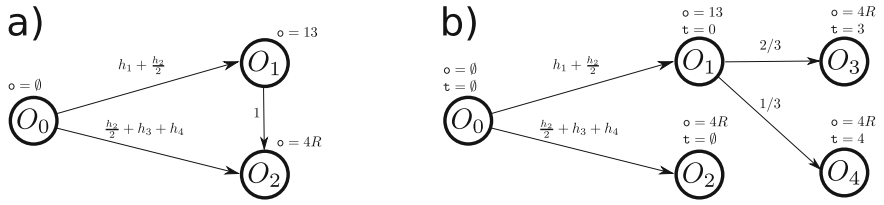


Fig. 13. Markov chains for Timed Onion Routing: a) $\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{N}}$ b) $\mathbb{C}/\mathcal{R}_{\mathcal{A}_\mathcal{T}}$

orthogonal idea and could be applied to our approach as well, substituting min-leakage with Shannon leakage.

The Lattice of Information approach to security seems to be related to the Abstract Interpretation approach to code obfuscation investigated by Giacobazzi et al. [23]; it would be interesting to further understand the connection between these approaches.

10 Conclusion

We presented a method to quantify the information leakage of a probabilistic system to an attacker. The method considers the probabilistic partial information semantics of the system and allows to encode attackers that can partially observe the internal behavior of the system. The method presented can be fully automated, and an implementation is being developed. The paper extends the consolidated LoI approach for leakage computation to programs with randomized behavior.

We extended the method to compute the channel capacity of a program, thus giving a security guarantee that does not depend on a given attacker, but considers the worst case scenario. We show how this can be obtained by maximizing an equation parameterized on the prior information of the attacker. The automatization of this computation raises interesting theoretical problems, as it requires to encode the property that all probability distributions on state must be derived from the same prior information, and thus involves a global constraint. We intend to work further on identifying suitable optimizations for constraints arising in this problem.

Finally, we analyzed the channel capacity of the Onion Routing protocol, encoding the classical attacker able to observe the traffic in a node and also a new attacker with time-tracking capacilities, and we proved that the time-tracking attacker is able to infer more information about the secret of the system.

References

1. Malacaria, P.: Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. CoRR abs/1101.3453 (2011)
2. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15, 321–371 (2007)

3. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: Gates, C., Franz, M., McDermott, J.P. (eds.) ACSAC, pp. 261–269. ACM (2010)
4. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. Inf. Comput. 206, 378–401 (2008)
5. Chen, H., Malacaria, P.: Quantifying maximal loss of anonymity in protocols. In: Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V. (eds.) ASIACCS, pp. 206–217. ACM (2009)
6. Köpf, B., Smith, G.: Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In: CSF, pp. 44–56. IEEE Computer Society (2010)
7. Köpf, B., Basin, D.A.: An information-theoretic model for adaptive side-channel attacks. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) ACM Conference on Computer and Communications Security, pp. 286–296. ACM (2007)
8. Millen, J.K.: Covert channel capacity. In: IEEE Symposium on Security and Privacy, pp. 60–66 (1987)
9. Goldschlag, D.M., Reed, M.G., Syverson, P.F.: Onion routing. Commun. ACM 42, 39–41 (1999)
10. Murdoch, S.J., Danezis, G.: Low-cost traffic analysis of tor. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, SP 2005, pp. 183–195. IEEE Computer Society, Washington, DC (2005)
11. Abbott, T.G., Lai, K.J., Lieberman, M.R., Price, E.C.: Browser-Based Attacks on Tor. In: Borisov, N., Golle, P. (eds.) PET 2007. LNCS, vol. 4776, pp. 184–199. Springer, Heidelberg (2007)
12. Cover, T., Thomas, J.: Elements of information theory. Wiley, New York (1991)
13. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal 27, 379–423 (1948)
14. Biondi, F., Legay, A., Nielsen, B.F., Wąsowski, A.: Maximizing entropy over markov processes (2012) (under review), http://www.itu.dk/people/fbio/maxent.pdf
15. Landauer, J., Redmond, T.: A lattice of information. In: CSFW, pp. 65–70 (1993)
16. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series. MIT Press (1993)
17. Malacaria, P.: Risk assessment of security threats for looping constructs. Journal of Computer Security 18, 191–228 (2010)
18. Malacaria, P., Chen, H.: Lagrange multipliers and maximum information leakage in different observational models. In: Erlingsson, Ã., Pistoia, M. (eds.) PLAS, pp. 135–146. ACM (2008)
19. Alvim, M.S., Andrés, M.E., Palamidessi, C.: Quantitative information flow in interactive systems. Journal of Computer Security 20, 3–50 (2012)
20. Chen, H., Malacaria, P.: The Optimum Leakage Principle for Analyzing Multithreaded Programs. In: Kurosawa, K. (ed.) ICITS 2009. LNCS, vol. 5973, pp. 177–193. Springer, Heidelberg (2010)
21. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic Quantification of Cache Side-Channels. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 564–580. Springer, Heidelberg (2012)
22. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: CSF (2012)
23. Preda, M.D., Giacobazzi, R.: Semantics-based code obfuscation by abstract interpretation. Journal of Computer Security 17, 855–908 (2009)

Reductions for Synthesis Procedures*

Swen Jacobs[1], Viktor Kuncak[2], and Philippe Suter[2]

[1] Graz University of Technology, Austria
swen.jacobs@iaik.tugraz.at
[2] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract. A synthesis procedure acts as a compiler for declarative specifications. It accepts a formula describing a relation between inputs and outputs, and generates a function implementing this relation. This paper presents the first synthesis procedures for 1) algebraic data types and 2) arrays. Our procedures are reductions that lift a synthesis procedure for the elements into synthesis procedures for containers storing these elements. We introduce a framework to describe synthesis procedures as systematic applications of inference rules. We show that, by interpreting both synthesis problems and programs as relations, we can derive and modularly prove widely applicable transformation rules, simplifying both the presentation and the correctness argument.

1 Introduction

Software synthesis is an active area of research [6,17,19,22]. It has received increased attention recently, but has been studied for decades [3,11,12,16]. Our paper pursues the synthesis of functions mapping inputs to outputs. The synthesized functions are guaranteed to satisfy a given input/output relation expressed in a decidable logic. We call this approach complete functional synthesis [8,9]. The appeal of this direction is that it can synthesize functions over unbounded domains, and that the produced code is guaranteed to satisfy the specification for the entire unbounded range of inputs. If the synthesis process always terminates, we speak of *synthesis procedures*, analogously to decision procedures.

Previous work described synthesis procedures for linear arithmetic and sets [8,9] as well as extensions to unbounded bitvector constraints [4,18]. In this paper we make further steps towards systematic derivation of synthesis procedures by showing how inference rules that describe decision procedure steps (possibly for a combination of theories) can be generalized to synthesis procedures. Within this framework we derive the first synthesis procedures for two relevant decidable theories of data structures: term algebras (algebraic data types), and the

* Swen Jacobs was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), the Austrian Science Fund (FWF) through the national research network RiSE (S11406), the Swiss NSF Grant 200021_132176, and COST Action IC0901. Philippe Suter was supported in part by the Swiss NSF Grant 200021_120433.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 88–107, 2013.

theory of integer-indexed arrays with symbolic bounds on index ranges. The two synthesis procedures that we present are interesting in their own right. Synthesis for algebraic data types can be viewed as a generalization of the compilation of pattern matching, and is therefore a useful way to increase the expressive power of functional programs. Synthesis for arrays is useful for synthesizing fragments of imperative programs. Synthesizing from constraints on arrays is challenging because it requires, in general, iteration over the space of indices. It therefore illustrates the importance of synthesizing not only individual values that meet a constraint, but also functions that enumerate all values.

Our synthesis procedures are expressed as a set of modular transformation rules whose correctness can be checked in a straightforward way, and which can be more easily implemented (even in foundational proof assistants). The transformations gradually evolve a constraint into a program. Sound rules for such transformations can be formulated for each decidable theory separately, and they can be interleaved for more efficient synthesis and more efficient synthesized programs. Our framework therefore contributes to the methodology for synthesis in general. We start from proof rules for a decision procedure, and extend them into transformation rules that can be viewed as a result of partially evaluating the execution of inference rules.

As remarked in [9], compiled synthesis procedures could be viewed as a result of partial evaluation of the execution of a constraint solver at run time. This is a useful observation from a methodological point of view. However, it likely has similar limitations as an attempt to automatically transform an interpreter into a compiler. We therefore expect that the insights of researchers will continue to play a key role in designing synthesis procedures. These insights both take the form of understanding decidable logics, but also understanding how to solve certain classes of problems efficiently. Examples of manually deriving compiled code that can be more efficient than run-time search appear in both synthesis for term algebras and the synthesis of arrays. We can assume that the values in these theories are finitely generated by terms. Because these terms become known only at run time, it appears, at first, necessary to continue running decision procedure at run time. However, because the nature of processing steps is known at compile time, it was possible to generate statically known loops instead of an invocation of a general-purpose constraint solver at run time. The main advantage is not only that such code can be more efficient by itself, but that it can then be further analyzed and simplified, automatically or manually, to obtain code that is close or better than one written using conventional development methodology.

Contributions. In summary, this paper makes the following contributions:

1. the first synthesis procedure for quantifier-free theory of algebraic data types;
2. the first synthesis procedure for a theory of (symbolically bounded) arrays;
3. a formalization of the above procedures, as well as a simple synthesis procedure for Presburger arithmetic, in a unified framework supporting:
   (a) proving correctness of synthesis steps, and
   (b) combining synthesis procedures in a sound way.

We start by introducing our framework and illustrate it with a simple synthesis procedure for Presburger arithmetic. We then present the synthesis procedures for algebraic data types and for arrays.

2 Synthesis Using Relation Transformations

A *synthesis problem* is a triple

$$[\![\bar{a}\ \ \langle\phi\rangle\ \ \bar{x}]\!]$$

where $\bar{a}$ is a set of *input* variables, $\bar{x}$ is a set of *output* variables and ϕ is a formula whose free variables are a subset of $\bar{a} \cup \bar{x}$. A synthesis problem denotes a binary relation $\{(\bar{a}, \bar{x}) \mid \phi\}$ between inputs and outputs. The goal of synthesis is to transform such relations until they become executable programs. Programs correspond to formulas of the form $P \wedge (\bar{x} = \bar{T})$ where $\mathrm{vars}(P) \cup \mathrm{vars}(\bar{T}) \subseteq \bar{a}$. We denote programs

$$\langle P \mid \bar{T} \rangle$$

We call the formula P a *precondition* and call the term $\bar{T}$ a *program term*.

We use $\vdash$ to denote the transformation on synthesis problems, so

$$[\![\bar{a}\ \ \langle\phi\rangle\ \ \bar{x}]\!] \vdash [\![\bar{a}\ \ \langle\phi'\rangle\ \ \bar{x}]\!] \tag{1}$$

means that the problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ can be transformed into the problem $[\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!]$. The variables on the right-hand side are always the same as on the left-hand side. Our goal is to compute, given $\bar{a}$, one value of $\bar{x}$ that satisfies ϕ. We therefore define the soundness of (1) as a process that refines the binary relation given by ϕ into a smaller relation given by ϕ', without reducing its domain. Expressed in terms of formulas, the conditions become the following:

$$\begin{array}{rl} \phi' \models \phi & \textit{refinement} \\ \exists \bar{x}.\phi \models \exists \bar{x}.\phi' & \textit{domain preservation} \end{array}$$

In other words, $\vdash$ denotes domain-preserving refinement of relations. Note that the dual entailment $\exists \bar{x}.\phi' \models \exists \bar{x}.\phi$ also holds, but it follows from *refinement*. Note as well that $\vdash$ is transitive.

Equivalences in the theory of interest immediately yield useful transformation rules: if ϕ and ϕ' are equivalent, (1) is sound. We can express fact as the following *inference rule*:

$$\frac{\models \phi_1 \leftrightarrow \phi_2}{[\![\bar{a}\ \ \langle\phi_1\rangle\ \ \bar{x}]\!] \vdash [\![\bar{a}\ \ \langle\phi_2\rangle\ \ \bar{x}]\!]} \tag{2}$$

In most cases we will consider transformations whose result is a program:

$$[\![\bar{a}\ \ \langle\phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle$$

The correctness of such transformations reduces to

$$\begin{array}{rl} P \models \phi[\bar{x} \mapsto \bar{T}] & \textit{refinement} \\ \exists \bar{x}.\phi \models P & \textit{domain preservation} \end{array}$$

A *synthesis procedure* for a theory $\mathcal{T}$ is given by a set of inference rules and a strategy for applying them such that every formula in the theory is transformed into a program.

2.1 Theory-Independent Inference Rules

We next introduce inference rules for a logic with equality. These rules are generally useful and are not restricted to a particular theory.

Equivalence. From the transitivity of $\vdash$ and (2), we can derive a rule for synthesizing programs from equivalent predicates.

$$\frac{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \models \phi_1 \leftrightarrow \phi_2}{[\![\bar{a}\ \langle\phi_2\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle}$$

Ground. In the case where no input variables are given, a synthesis problem is simply a satisfiability problem.

$$\frac{\mathcal{M} \models \phi}{[\![\emptyset\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle \top \mid \mathcal{M}\rangle} \qquad \frac{\neg\exists\mathcal{M}.\mathcal{M} \models \phi}{[\![\emptyset\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle \bot \mid \bot\rangle}$$

(In these rules $\mathcal{M}$ is a *model* for ϕ and should be thought of as a tuple of ground terms.) Note that the second rule can be generalized: even in the presence of input variables, if the synthesis predicate ϕ is unsatisfiable, then the generated program must be $\langle \bot \mid \bot\rangle$.

Assertions. Parts of a formula that only refer to input variables are essentially assertions and can be moved to the precondition.

$$\frac{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \mathrm{vars}(\phi_2) \subseteq \bar{a}}{[\![\bar{a}\ \langle\phi_1 \wedge \phi_2\rangle\ \bar{x}]\!] \vdash \langle \phi_2 \wedge P \mid \bar{T}\rangle}$$

Case Split. A top-level disjunction in the formula can be handled by deriving programs for both disjuncts and combining them with an if-then-else structure.

$$\frac{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash \langle P_1 \mid \bar{T}_1\rangle \qquad [\![\bar{a}\ \langle\phi_2\rangle\ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2\rangle}{[\![\bar{a}\ \langle\phi_1 \vee \phi_2\rangle\ \bar{x}]\!] \vdash \langle P_1 \vee P_2 \mid \mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\}\rangle}$$

Unconstrained Output. Output variables that are not constrained by ϕ can be assigned any value.

$$\frac{[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad x_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \langle\phi\rangle\ x_0\,;\bar{x}]\!] \vdash \langle P \mid \mathsf{any}\,;\bar{T}\rangle}$$

In the program, any denotes a nullary function that returns an arbitrary of the appropriate type.

One-point. Whenever the value of an output variable is uniquely determined by an equality atom, it can be eliminated by a simple substitution.

$$\frac{[\![\bar{a}\ \ \langle \phi[x_0 \mapsto t] \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(t)}{[\![\bar{a}\ \ \langle x_0 = t \wedge \phi \rangle\ \ x_0 \,;\, \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ \bar{x} := \bar{T}\ \mathsf{in}\ (t \,;\, \bar{x}) \rangle}$$

Definition. The definition rule is in a sense dual to One-point, and is convenient to give a name to a subterm appearing in a formula. Typical applications include purification and flattening of terms.

$$\frac{[\![\bar{a}\ \ \langle x_0 = t \wedge \phi[t \mapsto x_0] \rangle\ \ x_0 \,;\, \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (x_0 \,;\, \bar{x}) := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

Sequencing. The sequencing rule allows us to synthesize values for two groups of variables one after another. It fixes the values of some of the output variables, treating them temporarily as inputs, and then continues with the synthesis of the remaining ones.

$$\frac{[\![\bar{a} \,;\, \bar{x}\ \ \langle \phi \rangle\ \ \bar{y}]\!] \vdash \langle P_1 \mid \bar{T}_1 \rangle \qquad [\![\bar{a}\ \ \langle P_1 \rangle\ \ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2 \rangle}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x} \,;\, \bar{y}]\!] \vdash \langle P_2 \mid \mathsf{let}\ \bar{x} := \bar{T}_2\ \mathsf{in}\ (\bar{x} \,;\, \bar{T}_1) \rangle}$$

Static Computation. A basic rule is to perform computational steps when possible.

$$\frac{[\![a_0 \,;\, \bar{a}\ \ \langle \phi[t \mapsto a_0] \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad \mathrm{vars}(t) \subseteq \bar{a} \qquad a_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle \mathsf{let}\ a_0 := t\ \mathsf{in}\ P \mid \mathsf{let}\ a_0 := t\ \mathsf{in}\ \bar{T} \rangle}$$

Variable Transformation. The $\vdash$ transformation preserves the variables. To show how we can change the set of variables soundly, we next present in our framework *variable transformation* by a computable function ρ [8], as an inference rule on two $\vdash$ transformations.

$$\frac{[\![\bar{a}\ \ \langle \phi[\bar{x} \mapsto \rho(\bar{x}')] \rangle\ \ \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \rho(\bar{T}) \rangle}$$

Slightly more generally, we have the following:

$$\frac{[\![\bar{a}\ \ \langle \phi' \rangle\ \ \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle \qquad \exists \bar{x}.\phi \models \exists \bar{x}'.\phi' \qquad \phi' \models \phi[\bar{x} \mapsto \rho(\bar{x}')]}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \rho(\bar{T}) \rangle}$$

Existential Projection. This rule is a special case of variable transformation, where ρ simply projects out some of the variables.

$$\frac{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x} \,;\, \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \ \langle \exists \bar{x}'.\phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (\bar{x} \,;\, \bar{x}') := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

3 Synthesis for Presburger Arithmetic

This section summarizes a simple version of a synthesis procedure for Presburger arithmetic using our current synthesis rules. Our goal is to give a complete procedure that is easy to prove correct, as opposed to one that generates efficient code. The reader will observe that our description reads like a description of quantifier elimination. Note, however, that the inference rules that we refer to are from the previous section and therefore also specify how to compute the corresponding program.

Unlike the procedure in [9] the procedure below does not perform efficient solving of equations, but could be refined to do so by adapting the description in [9] to our inference rules.

As in the preprocessing steps for simple quantifier elimination for Presburger arithmetic, the equivalences we use as rules include replacing $t_1 \neq t_2$ with $t_1 < t_2 \vee t_2 < t_1$. In principle, we can rewrite $t_1 = t_2$ into $t_1 \leq t_2 \wedge t_2 \leq t_1$ (see [9] for more efficient approaches). We rewrite $t_1 \leq t_2$ into $t_1 < t_2 + 1$. When needed, we assume that we apply the Case Split rule to obtain only a conjunction of literals. We also assume that we apply the Sequencing rule to fix the remaining variables and only consider one output variable x. Finally, thanks to the Assertions rule, we assume that all literals contain x.

A rule that takes into account divisibility is the following:

$$\frac{[\![\bar{a}\ \langle \phi[kx \mapsto y] \wedge y \equiv_k 0 \rangle\ y]\!] \vdash \langle P \mid T \rangle \qquad k \neq 0 \qquad x \text{ in } \phi \text{ only as } kx}{[\![\bar{a}\ \langle \phi \rangle\ x]\!] \vdash \langle P \mid T/k \rangle}$$

The rule is a case of Variable Transformation with $\rho(y) = y/k$.

To enable the previous rule, we can ensure that all occurrences of a variable have the same coefficient by multiplying constraints by a positive constant (e.g., the least common multiple of all coefficients). These transformations are based on using (in a context) equivalences between $t_1 \bowtie t_2$ and $kt_1 \bowtie kt_2$, for $k > 0$ and $\bowtie \in \{=, <, >, \equiv_p\}$.

Using the rules so far, we can ensure that an output variable has a coefficient 1. If such a variable occurs in at least one equality, we can eliminate it using One-point. If the variable occurs only in inequalities, we perform the main step of the procedure.

Elimination of Inequalities. Based on the discussion above, we can assume that the formula ϕ in the synthesis problem is of the form

$$\bigwedge_{i=1}^{L} l_i < x \wedge \bigwedge_{j=1}^{U} x < u_j \wedge \bigwedge_{i=1}^{D} x + t_i \equiv_{K_i} 0$$

We aim to replace ϕ with ϕ' such that

$$[\![\bar{a}\ \langle \phi \rangle\ x]\!] \vdash [\![\bar{a}\ \langle \phi' \rangle\ x]\!] \tag{3}$$

We define ϕ' as

$$\bigvee_{i=1}^{L} \bigvee_{k=1}^{K} (\phi \wedge x = l_i + k)$$

where K is the least common multiple of $K_1, \ldots, K_D$. Clearly ϕ' is stronger than ϕ because each disjunct is stronger than ϕ, so it remains to argue about domain preservation. Suppose there exists values for x so that ϕ holds. Let l_I be the largest value among the values of lower bounds l_i and let T be such that $l_I + T \equiv_K x$ holds. Then letting x to be $l_I + T$ makes ϕ' true as well.

After performing disjunctive splitting (Case Split), we can eliminate x using the One-point rule. The correctness follows by (3) and the correctness of One-point. The cases where some of the bounds do not exist can be treated similarly.

This completes the overview of synthesis of functions given by Presburger arithmetic relations.

Enumerating Solutions. In addition to finding one solution $\bar{x}$ such that ϕ holds, it is useful to be able to find all solutions, when this set is finite. When solving constraints at run time, a simple way to find all solutions is to maintain a list of previously found solutions $\bar{v}_1, \ldots, \bar{v}_n$ for $\bar{x}$ and add to ϕ an additional conjunct $\bigwedge_{i=1}^{n} \bar{x} \neq \bar{v}_n$, see [7].

One possible approach to compile this process is to enrich Presburger arithmetic with finite uninterpreted relations as parameters. This enables a synthesis procedure to, for example, take the set of previous solutions as the input. If R is such a finite-relation symbol or arity n and $\bar{x}$ are n variables, we introduce an additional literal $\bar{x} \notin R$ into the logic, with the intention that R stores the previously found solutions. The elimination of inequalities then produces terms that avoid the elements of R by considering not only the value $l_i + k$ for x, but enumerating a larger number of solutions, $l_i + k + \alpha N$, for multiple values of $\alpha \geq 0$. Because R is known only at run time, the generated code contains a loop that increases $\alpha \geq 0$ to allow x to leave the range of the corresponding coordinate of R. The value of α is bounded at run time by, for example, $\lceil (\max(R_x) - \min(R_x))/K \rceil + 1$ where R_x is the projection of R onto the coordinate at which x appears in the literal $\bar{x} \notin R$. The generated loop is guaranteed to terminate.

4 Synthesis for Term Algebras

This section presents a synthesis procedure for quantifier-free formulas in the theory of term algebras. We start by assuming a pure term algebra, and later extend the system to algebras with elements from parameter theories. In both cases, we present a series of normal forms and inference rules, and argue that together they can be used to build a synthesis procedure.

4.1 Pure Term Algebras

The grammar of atoms over our term algebra is given by the following two production rules, where c and F denote a constant and a function symbol from the algebraic signature, respectively:

$$A ::= T = T \mid T \neq T \mid \mathrm{is}_c(T) \mid \mathrm{is}_F(T)$$
$$T ::= x \mid c \mid F(\bar{T}) \mid F_i(T)$$

In the following we assume that the algebra defines at least one constant and one non-nullary constructor function. Formulas are built from atoms with the usual propositional connectives. We use an extension of the standard theory of term algebras. The extension defines additional unary *tester* functions $\mathrm{is}_c(\cdot)$ and $\mathrm{is}_F(\cdot)$ for constant and functions in the algebraic signature respectively, and unary *selector* functions $F_i(\cdot)$, with $1 \leq i \leq n$ where n is the arity of F. These extra symbols form a definitional extension [5] given by the axioms:

$$\forall x.\mathrm{is}_c(x) \leftrightarrow x = c \tag{4}$$
$$\forall x.\mathrm{is}_F(x) \leftrightarrow \exists \bar{y}.x = F(\bar{y}) \tag{5}$$
$$\forall x, y.F_i(x) = y \leftrightarrow (\exists \bar{y}.y = \bar{y}\,[i] \wedge F(\bar{y}) = x) \vee \neg(\exists \bar{y}.F(\bar{y}) = x) \wedge x = y \tag{6}$$

Note that the case analysis in (6) is required only to make the selector functions total. In practice, we are only interested in cases where the selectors are applied to arguments of the proper type. We will therefore assume in the following that each selector application $F_i(x)$ is accompanied with a side condition $\mathrm{is}_F(x)$.

Rewriting of tester and selector functions. By applying the axioms (4) and (5), we can rewrite all applications of a tester function into an existentially quantified equality over terms. We can similarly eliminate applications of testers by existentially quantifying over the arguments of the corresponding constructors. Using the Existential Projection rule, we can in turn consider the obtained synthesis problem as a quantifier-free one.

Elimination through unification. We can at any point apply *unification* to a conjunction of equalities over terms. Unification rewrites a conjunction of term equations into either $\bot$, if the equations contain a cycle or an equality involving incompatible constructors, or into an equivalent conjunction of atoms $\bigwedge_i v_i = t_i$, where v_i is a variable and t_i is a term. This set of equations has the additional property that

$$\left(\bigcup_i \{v_i\}\right) \cap \left(\bigcup_i \mathrm{vars}(t_i)\right) = \emptyset$$

In other words, it defines a set of variables as a set of terms built from another disjoint set of variables [2]. This form is particularly suitable for applications of the One-point rule: indeed, whenever v_i is an output variable, we can apply it, knowing that v_i does not appear in t_i (or in any other equation).

Dual view. Unification allows us to eliminate output variables that are to the left of an equality. When instead an input variable appears in such position, we can resort to a dual form to eliminate output variables appearing in the right-hand side. We obtain the dual form by applying as much as possible the following two rules to term equalities:

$$\frac{t = c}{\mathrm{is}_c(t)} \qquad \frac{t = F(t_1, \ldots, t_n)}{F_1(t) = t_1 \wedge \ldots \wedge F_n(t) = t_n \wedge \mathrm{is}_F(t)}$$

Note that these are rewrite rules for formulas. Because they preserve the set of variables and equisatisfiability, they can be lifted to inference rules for programs using the Equivalence rule. Observe that at saturation, the generated atoms are of two kinds: 1) applications of tester predicates and 2) equalities between two terms, each containing at most one variable. In particular, all equalities between an output variable and a term are amenable to applications of the One-point rule.

Disequalities. Finally, we introduce a dedicated rule for the treatment of disequalities between terms. The rule is defined for disequalities over variables and constants in *conjunctive normal form* (CNF). From a conjunction of disequalities over terms, we can obtain CNF by applying the following rewrite rules until saturation:

$$\frac{F(\bar{t}_1) \neq G(\bar{t}_2) \qquad F \neq G}{\top} \qquad \frac{F(\bar{t}_1) \neq F(\bar{t}_2)}{t_1^1 \neq t_1^n \vee \ldots \vee t_2^1 \neq t_2^n}$$

Intuitively, the first rule captures the fact that terms built with distinct constructors are trivially distinct (note that this also captures distinct constants, which are nullary constructors). The second rule breaks down a disequality into a disjunction of disequalities over subterms.

To obtain witness terms from the CNF, it suffices to satisfy one disequality in each conjunct. We achieve this by eliminating one variable after another, applying for each a diagonalization principle, as follows. In the following rule ϕ_{CNF} denotes the part of the CNF formula over atomic disequalities that does not contain a given variable of interest x_0.

$$\frac{[\![\bar{a}\ \langle \phi_{\mathrm{CNF}} \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \phi_{\mathrm{CNF}}}{\left[\!\!\left[\bar{a} \left\langle \begin{array}{c} (x_0 \neq t_1 \vee \ldots) \\ \wedge \quad \ldots \\ \wedge\ (x_0 \neq t_n \vee \ldots) \\ \wedge \quad \phi_{\mathrm{CNF}} \end{array} \right\rangle x_0 \,;\, \bar{x} \right]\!\!\right] \vdash \langle P \mid \mathsf{let}\ \bar{x} := \bar{T}\ \mathsf{in}\ (\Delta(t_1, \ldots, t_n)\,;\, \bar{T}) \rangle}$$

In the generated program, Δ denotes an n-ary computable function that returns, at run time, a term distinct from all its arguments. Such a value is guaranteed to exist, since the term algebra is assumed to have at least one constructor. This function runs in time polynomial in the number of its arguments.

Synthesis Procedure for Algebraic Data Types. We now argue that the reductions to normal forms and the rules presented above are sufficient to form a complete synthesis procedure for any given pure term algebra structure. The procedure (a strategy for applying the rules) is given by the following steps:

1. Reduce an arbitrary propositional structure to a conjunction through applications of the Case Split rule.
2. Remove selectors and testers through rewrites and applications of Existential Projection.
3. Apply unification to all equalities, then apply One-point as often as possible. As a result, the only equalities remaining have the form $a = t$, where a is an input variable and $a \notin \text{vars}(t)$.
4. Rewrite into dual form, then apply One-point as much as possible. After applying Assertions, the problem is reduced to a conjunction of disequalities, each involving at least one output variable.
5. Transform the conjunction into CNF and eliminate all remaining variables by successive applications of the diagonalization rule.

Given a conjunctions of literals, the generated program runs in time polynomial in the size of the input terms: it consists of a sequence of assignments, one for each output variable, and each term has polynomial size.

4.2 Reduction to an Interpreted Theory

We now consider the case of a term algebra defined over an interpreted theory $\mathcal{T}$. A canonical example is the algebra of integer lists, where $\mathcal{T}$ is the theory of integers, and defined by the constant $\mathsf{Nil} : \mathsf{List}$ and the constructor $\mathsf{Cons} : \mathbb{Z} \times \mathsf{List} \to \mathsf{List}$. In this theory, the selector function $\mathsf{Cons}_1(\cdot)$, for instance, is of type $\mathsf{List} \to \mathbb{Z}$. We show how to reduce a synthesis problem in the combination of theories to a synthesis problem in $\mathcal{T}$. We focus on the important differences with the previous case.

Purification. We can assume without loss of generality that constructor terms contain no subterms from $\mathcal{T}$ other than variables. Indeed one can always apply the Definition rule to separate such terms.

Unification. Applying unification can result in derived equalities between variables of $\mathcal{T}$. These should simply be preserved in the reduced problem.

Dual view. Applying the rewriting into the dual view can result in derived equalities of the form $x = t$, where x is a variable from $\mathcal{T}$ and t is an application of selectors to an input variable. Because $\mathcal{T}$ cannot handle these selectors, we need to rewrite t into a simple variable. By using the Definition and Sequencing rules, we make this variable an input of the problem in $\mathcal{T}$.

Disequalities. Contrary to the pure case, we cannot always eliminate all conjuncts in the CNF by applying a diagonalization; we can eliminate variables that belong to the term algebra, but not variables of $\mathcal{T}$. Instead, for each disequality $v \neq w$ over $\mathcal{T}$ in the CNF, we introduce at the top-level a disjunction $v = w \vee v \neq w$, and apply the Case Split rule to encode a guess. This in essence compiles the guessing of the partitioning of shared variables that is traditionally introduced in a Nelson-Oppen setting [15]. Because Case Split preserves the relation entirely, this is a sound and complete reduction step.

Once all the disequalities have been handled, either through diagonalization if they are over algebraic terms, or by case-splitting if they are over $\mathcal{T}$ variables, we have entirely reduced the synthesis problem into a synthesis problem for $\mathcal{T}$.

5 Synthesis for Arrays with Symbolic Bounds

This section introduces a synthesis procedure for a theory of arrays. In contrast to many other theories for which synthesis procedures have been introduced, the standard (unbounded) array theory does not admit quantifier elimination. With a known finite bound on the size of all arrays, there is a procedure that reduces the synthesis problem to synthesis problems over indices and elements, in a similar way as the satisfiability problem for arrays is reduced to these component theories. However, if we do not know the size bounds at compile time, we need to employ a mixed approach, which postpones some of the reasoning to run time. The reduction is the same as before, but now the component synthesis procedures not only return one solution of the synthesis problem, but instead an iterator over all possible solutions (given by any limited knowledge about the inputs contained in the specification formula). Then, at run time, the synthesized code examines all the solutions for constraints on indices, searching for one that matches the current array inputs.

In the rest of this section we focus on this more general case of symbolic bounds, then revisit briefly statically bounded arrays as a special case.

5.1 Preliminaries

We present synthesis for a *theory of arrays with symbolic bounds*. We consider arrays with the usual read and write operations, an index theory $\mathcal{T}_I$ with an ordered, discrete domain, and an element theory $\mathcal{T}_E$. We assume that our input formula ϕ is a conjunction of literals, and that we have synthesis procedures for these theories. Additionally, we assume that we have a predicate $\approx_I$ between arrays, where I can be any set of variables or constants, and $a \approx_I b$ evaluates to true iff a and b are equal up to (the elements stored at) indices $i \in I$. In particular, this also subsumes extensionality of arrays (with $I = \emptyset$).

Arrays with Symbolic Bounds. We assume that our specification ϕ contains, for every array variable a, two special variables a_l, a_u, standing for the lower and upper bound of the array. Additionally, we assume that ϕ contains, for every

index variable i used to read or write into a, the constraints $a_l \leq i \wedge i \leq a_u$. These constraints ensure that synthesized programs to not contain out-of-bounds array accesses, and that the number of possible solutions for index variables will be bounded at run time. If a is an array parameter, then a_l, a_u are additional parameters. For convenience, if we have $b = \mathsf{write}(a, i, e)$ or $a \approx_I b$, then we assume that $a_l = b_l$ and $a_u = b_u$, i.e. we need not introduce multiple lower and upper bounds for "connected" arrays.

Enumerating Solutions For the index theory $\mathcal{T}_I$, we need a synthesis procedure that not only returns one solution $\overline{T}$, but allows us to iterate over the set $\overline{T}^*$ of all possible solutions. We assume that one of the following cases holds:

1. the synthesis procedure computes $\overline{T}^*$ as a finite set of solutions
2. the synthesis procedure computes $\overline{T}^*$ as a solved form of ϕ, that allows us to access solutions iteratively (like mentioned in Sect. 3; this is also possible if there are infinitely many possible solutions at compile time)
3. the synthesis procedure produces code $\overline{T}^*$, representing a specialized solver for the index theory, that is instantiated with ϕ and allows to add more constraints ψ to obtain solutions satisfying $\phi \wedge \psi$ (in the limit this means integrating a constraint solver procedure into the generated code [7]).

5.2 A Reduction-Based Synthesis Procedure for Arrays

We introduce a synthesis procedure for arrays, consisting of the following steps:

- **Array reduction:** ϕ is reduced to $\mathcal{T}_E \cup \mathcal{T}_I$, along with a set of definitions that allows us to generate witness terms for array variables;
- **"Partial Synthesis" reduction:** part of the reasoning is postponed to runtime, assuming we get an enumerator of all possible solutions in $\mathcal{T}_E \cup \mathcal{T}_I$;
- **Separation and synthesis in $\mathcal{T}_E$ and $\mathcal{T}_I$:** we separate the specification into parts talking purely about $\mathcal{T}_E$ and $\mathcal{T}_I$, and synthesize all possible solutions.

Array Reduction. We introduce fresh variables for array writes and reads, allowing us to reduce the problem to the combined theory $\mathcal{T}_I \cup \mathcal{T}_E$:

1. For every array write $\mathsf{write}(a, i, e)$ in ϕ: i) use Definition to introduce a fresh array variable b, and obtain $b = \mathsf{write}(a, i, e) \wedge \phi[write(a, i, e) \mapsto b]$, and ii) by Equivalence, add $b[i] = e \wedge a \approx_{\{i\}} b$ to ϕ.
2. Until saturation, use Equivalence to add for every pair of literals $a \approx_I b$ and $b \approx_J c$ in ϕ the literal $a \approx_{I \cup J} c$.
3. For every array read $a[i]$ and predicate $a \approx_J b$ in ϕ: use Equivalence to add a formula $(\bigwedge_{j \in J} i \neq j) \rightarrow a[i] = b[i]$ to ϕ.
4. For every array read $a[i]$ in ϕ: i) use Definition to introduce a fresh element variable a_i , and obtain $a_i = a[i] \wedge \phi[a[i] \mapsto a_i]$.
5. For every pair of variables a_i, a_j in ϕ: use Equivalence to add a formula $i = j \rightarrow a_i = a_j$ to ϕ.

Let $D \equiv D_1 \wedge D_2$, where D_1, D_2 are the sets of definitions introduced in 1 and 4, respectively. Let Eq_A be the saturated set of all literals $a \approx_I b$ after 2, Impl the set of all implications introduced in 3 and 5, and ϕ' the remaining part of ϕ, after the rewriting steps in 1 and 5. Let furthermore $\bar{b}, \bar{a_i}$ be the sets of fresh variables introduced in steps 1 and 5, respectively. Then array reduction can be depicted as a macro-step

$$\frac{[\![\bar{a}\ \ \langle \phi' \wedge \mathsf{Impl} \wedge Eq_A \wedge D \rangle\ \ \bar{b}\,;\bar{a_i}\,;\bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (\bar{b}\,;\bar{a_i}\,;\bar{x}) := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

Let $\bar{x_A}\,;\bar{x_E}\,;\bar{x_I}$ be a separation of $\bar{x}$ into array, element and index variables. As an array-specific step, based on $Eq_A \wedge D$ we can now already give witness terms for array variables $\bar{b}\,;\bar{x_A}$, assuming that we will get witness terms $\bar{T}_{a_i}$ for $\bar{a_i}$ and $\bar{T}_I$ for $\bar{x_I}$:

Let a be an array variable, and I the set of all index variables i for which $a_i = a[i]$ is in D. For a given v, let J be the maximal subset of I s.t. $\forall i, j \in J.\ v \models i \neq j$. By construction, there must be an array variable b s.t. $a \approx_I b$ is in Eq_A. If b is not a parameter array, all positions not explicitly defined can be defined arbitrarily. Then the witness term T_a for variable a is defined by:

$$T_a := \mathsf{write}(\ldots(\mathsf{write}(b, T_{j_1}, T_{a_{j_1}})\ldots), T_{j_n}, T_{a_{j_n}})$$

where the T_j are witness terms for index variables $j \in J$, and the T_{a_j} witness terms for the corresponding element variables. Let $\bar{T_A}$ be the sequence of witness terms for all array variables $\bar{b}\,;\bar{x_A}$. Then this step can be depicted as

$$\frac{[\![\bar{a}\ \ \langle \phi' \wedge \mathsf{Impl} \wedge D_2 \rangle\ \ \bar{a_i}\,;\bar{x_E}\,;\bar{x_I}]\!] \vdash \langle P \mid \bar{T} \rangle}{\begin{array}{l}[\![\bar{a}\ \ \langle \phi' \wedge \mathsf{Impl} \wedge Eq_A \wedge D \rangle\ \ \bar{b}\,;\bar{x_A}\,;\bar{a_i}\,;\bar{x_E}\,;\bar{x_I}]\!] \\ \qquad \vdash \langle P \mid \mathsf{let}\ (\bar{a_i}\,;\bar{x_E}\,;\bar{x_I}) := \bar{T}\ \mathsf{in}\ (\bar{T_A}\,;\bar{a_i}\,;\bar{x_E}\,;\bar{x_I}) \rangle\end{array}}$$

Correctness of this step follows from the correctness of array decision procedures using the same reduction. Note that we also remove Eq_A and D_1 from our specification, as they will not be needed anymore.

Partial Synthesis with Run-Time Checks. Since the theory of arrays does not allow quantifier elimination, we in general need to postpone some of the reasoning to run time. The following is a general rule to separate the specification into a part ϕ that allows for compile-time synthesis, and another part ψ that is checked (against the possible solutions of ϕ) at run time. Here, we assume that the result $\bar{T}^*$ is an iterator over all possible solutions, and that for any given $\bar{a}$ only finitely many solutions exist:

$$\frac{[\![\bar{a}\ \ \langle\phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid \bar{T}^*\rangle}{[\![\bar{a}\ \ \langle\phi \wedge \psi\rangle\ \ \bar{x}]\!] \vdash \left\langle P \wedge \exists i.\ \psi[\bar{x} \mapsto \bar{T}^*.\mathsf{next}^{(i)}] \;\middle|\; \begin{array}{l} \bar{x} := \bar{T}^*; \\ \mathsf{while}(\neg\psi(\bar{a}, \bar{x})) \\ \mathsf{if}(\bar{T}^*.\mathsf{hasNext})\ \{\bar{x} := \bar{T}^*.\mathsf{next}\} \\ \mathsf{else}\ \{\mathsf{return\ UNSAT}\} \end{array} \right\rangle}$$

If $\bar{T}^*$ is not an iterator, but a specialized decision procedure for the given theory and constraint ϕ, then the loop is replaced by a call to $\bar{T}^*$ with constraint ψ.

For array synthesis, we apply the rule to remove D_2, reducing the synthesis problem to $[\![\bar{a}\ \ \langle\phi' \wedge \mathsf{Impl}\rangle\ \ \bar{a_i}\,;\bar{x_E}\,;\bar{x_I}]\!] \vdash \langle P \mid \bar{T}^*\rangle$, in the theory $\mathcal{T}_E \cup \mathcal{T}_I$.

Separation of $\mathcal{T}_I \cup \mathcal{T}_E$ We use the Sequencing rule to separately synthesize element and index variables:[1]

$$\frac{[\![\bar{a}\,;\bar{x_I}\ \ \langle\phi' \wedge \mathsf{Impl}\rangle\ \ \bar{a_i}\,;\bar{x_E}]\!] \vdash \langle P_E \mid \bar{T}_E\rangle \qquad [\![\bar{a}\ \ \langle P_E\rangle\ \ \bar{x_I}]\!] \vdash \langle P \mid \bar{T}_I^*\rangle}{[\![\bar{a}\ \ \langle\phi' \wedge \mathsf{Impl}\rangle\ \ \bar{x}\,;\bar{y}]\!] \vdash \langle P \mid \mathsf{let}\ \bar{x_I} := \bar{T}_I^*\ \mathsf{in}\ (\bar{T}_E\,;\bar{x_I})\rangle}$$

$\mathcal{T}_E$-Synthesis To solve the left-hand side, note that ϕ' is a conjunction of literals and does not contain array variables anymore, so it can be separated into $\phi_E \wedge \phi_I$, with ϕ_E, ϕ_I pure constraints in $\mathcal{T}_E$ and $\mathcal{T}_I$, respectively. We use the Assertions rule to move ϕ_I into P_E. The only other occurrences of index variables are in Impl. To remove these, we use Equivalence to introduce a disjunction over all possible valuations of equalities between index variables, and Case Split to branch synthesis of element variables for all these cases:

Let Eq_I be the set of all equalities $i = j$ s.t. either $i = j$ or $i \neq j$ appears in an implication in Impl. Let V_E be the set of truth valuations of elements of Eq_I, each described by a conjunction of literals $v \in V_E$ (containing for every $l \in Eq_I$ either l or $\neg l$). For every $v \in V_E$, obtain a new formula ϕ_v by adding to $\phi_E \wedge v$ the succedent of all implications in Impl for which the antecedent is in v.

For each $v \in V_E$, we solve $[\![\bar{a}\,;\bar{x_I}\ \ \langle\phi_v\rangle\ \ \bar{a_I}\,;\bar{x_E}]\!] \vdash \langle P_v \mid \bar{T}_v^*\rangle$, where ϕ_v is a pure $\mathcal{T}_E$-constraint. Joining results according to Case Split, we get $\langle P_E \mid \bar{T}^*\rangle$.

$\mathcal{T}_I$-Synthesis We solve the right-hand side of the Sequencing rule, $[\![\bar{a}\ \ \langle P_E\rangle\ \ \bar{x_I}]\!] \vdash \langle P \mid \bar{T}_I^*\rangle$. As before we use Assertions to obtain a pure $\mathcal{T}_I$-constraint.[2]

[1] Note that we do not have to explicitly compute all possible solutions in $\mathcal{T}_E$; since $\bar{x_I}$ is used as an input in $\bar{T}_E$, we will obtain a suitable solution of $\bar{a_i}\,;\bar{x_E}$ for every solution of $\bar{x_I}$.

[2] If bounds for all arrays (or all array indices in ϕ) can be computed at compile time, then all solutions can be computed statically. Otherwise, array bounds are symbolic and will only have values at run time, i.e. we need to be able to compute solutions during runtime.

Remarks For efficiency, it may be useful to deduce, both in $\mathcal{T}_I$ and $\mathcal{T}_E$, equalities that are implied by ϕ at any time, and replacing clauses in Impl by their succedents if the antecedent is implied by ϕ, or the negation of the antecedent, if the negation of the succedent is implied. This will avoid unnecessary branching, speeding up synthesis and removing dead branches from the resulting code.

Theory Combination $\mathcal{T}_E \cup \mathcal{T}_I$ The reduction above assumes that theories $\mathcal{T}_E$ and $\mathcal{T}_I$ are *strongly disjoint*, i.e. they share not even the equality symbol. Alternatively, we can make the restriction that variables that are used for array reads may never be compared to variables that are used as elements. In this case, implications from congruence of array reads is the only connection between the theories, and $\mathcal{T}_I$-synthesis can run completely independent of $\mathcal{T}_E$-synthesis, provided the latter accounts for all possible cases of $\mathcal{T}_I$-equalities. If the theories are not strongly disjoint, we really need a synthesis procedure for the combined theory. In this case, we directly use the combined decision procedure to produce an iterator over all possible solutions in both element and index theory.

Statically Bounded Arrays If all arrays in ϕ are statically bounded, i.e. values of upper and lower bounds are known or can be computed at compile time, then we can statically compute all solutions for constraints in $\mathcal{T}_I$ that are within array bounds. In that case the generated code does not need an iterator that computes additional solutions, and we can give a constant bound on the maximum number of traversals of the loop at compile time.

5.3 Complexity of Synthesis and Synthesized Code

Complexity of the array synthesis procedure is dominated by the branching on equalities of index variables: we may need exponentially many (in the number of index variables) calls to the synthesis procedure for $\mathcal{T}_E$. The array reduction itself runs in polynomial time.

Correspondingly, the size of the synthesized code is also exponential in the number of index variables. The code can contain branches for all possible cases of equalities (arrangements) among indices. Although only one of these branches will be explored at runtime, the worst-case running time of the synthesized code will still be exponential in the number of index variables: for a size bound n on a given array, there may be n^i many solutions to the constraints in $\mathcal{T}_I$. In the worst case, the condition of the while-loop needs to be checked for all of these solutions.

5.4 Example of Array Synthesis

Suppose we want to synthesize a most general precondition P and program code s.t. for any input array a and bounds a_l, a_u that satisfy P, the synthesized code

computes values for an array b and integer variables i, j, k such that the following is satisfied:[3]

$$\begin{aligned}\phi \equiv \quad & a_l = 0 \wedge i > a_l \wedge i = j + j \wedge i < a_u \wedge k \geq a_l \wedge k < i \\ & \wedge\, a[i] > 0 \wedge a[k] \leq 0 \wedge b[i] > a[i-2] \wedge b[k] = a[i] \\ & \wedge\, a' = \mathsf{write}(a, i, e_1) \wedge b = \mathsf{write}(a', k, e_2).\end{aligned}$$

Array reduction: We obtain

$$\begin{aligned}D_1 :=\ & \{a' = \mathsf{write}(a, i, e_1), b = \mathsf{write}(a', k, e_2)\} \\ Eq_A :=\ & \{a' \approx_i a, b \approx_k a', a \approx_{\{i,k\}} b\} \\ \mathsf{Impl} =\ & \left\{ \begin{array}{llll} k \neq i \ \rightarrow\ a_k = a'_k, & i-2 \neq i \ \rightarrow\ a_{i-2} = a'_{i-2}, \\ i \neq k \ \rightarrow\ a'_i = b_i, & i-2 \neq k \ \rightarrow\ a'_{i-2} = b_{i-2}, \\ i-2 \neq i \wedge i-2 \neq k \ \rightarrow\ a_{i-2} = b_{i-2} & \\ i = k \ \rightarrow\ a_i = a_k, & i-2 = k \ \rightarrow\ a_{i-2} = a_k, \\ i = k \ \rightarrow\ a'_i = a'_k, & i-2 = k \ \rightarrow\ a'_{i-2} = a'_k, \\ i = k \ \rightarrow\ b_i = b_k, & i-2 = k \ \rightarrow\ b_{i-2} = b_k \end{array} \right\}, \\ D_2 :=\ & \left\{ \begin{array}{l} a_i = a[i], a_{i-2} = a[i-2], a_k = a[k], \\ a'_i = a'[i], a'_{i-2} = a'[i-2], a'_k = a'[k], \\ b_i = b[i], b_{i-2} = b[i-2], b_k = b[k] \end{array} \right\} \\ \phi' :=\ & a_l = 0 \wedge i > a_l \wedge i = j + j \wedge i < a_u \wedge k \geq a_l \wedge k < i \\ & \wedge\, a_i > 0 \wedge a_k \leq 0 \wedge b_i > a_{i-2} \wedge b_k = a_i \\ & \wedge\, a'_i = e_1 \wedge b_k = e_2\end{aligned}$$

Implied equalities and disequalities: From ϕ' we can conclude that $i \neq j, k \neq i$ and $i - 2 \neq i$, as well as $a_i \neq a_k, b_i \neq a_{i-2}$.

Propagating equalities through Impl*:* $k \neq i$ implies $a_k = a'_k$ and $a'_i = b_i$. $i - 2 \neq i$ implies $a_{i-2} = a'_{i-2}$. In the opposite direction, $a_i \neq a_k$ implies $i \neq k$ (which we already knew). We get

$$\phi'' := \phi' \wedge a_i \neq a_k \wedge b_i \neq a_{i-2} \wedge a'_i = b_i \wedge a_{i-2} = a'_{i-2} \wedge i \neq k \wedge i - 2 \neq i.$$

Separation of $\mathcal{T}_I \cup \mathcal{T}_E$ We use the $\mathsf{Sequencing}$ rule, obtaining subproblems $[\![\bar{a}\,; \bar{x_I}\ \ \langle \phi'' \wedge \mathsf{Impl} \rangle\ \ \bar{a_i}\,; \bar{x_E}]\!] \vdash \langle P_E \mid \bar{T}_E \rangle$ and $[\![\bar{a}\ \ \langle P_E \rangle\ \ \bar{x_I}]\!] \vdash \langle P \mid \bar{T}_I^* \rangle$.

$\mathcal{T}_E$-Synthesis From the three equations that appear in antecedents of Impl, valuations for two are fixed by ϕ''. Thus, we only branch on the valuation of $i - 2 = k$. Let $v_1 \equiv i - 2 = k$, which implies $a_{i-2} = a_k, a'_{i-2} = a'_k$ and $b_{i-2} = b_k$. Let $v_2 \equiv i - 2 \neq k$, which implies $a'_{i-2} = b_{i-2}$ and (together with $i - 2 \neq i$) $a_{i-2} = b_{i-2}$.

[3] Note that the last two literals imply $a \approx_{\{i,k\}} b$, which in turn implies that there exist valuations for a', e_1, e_2 satisfying these literals. Thus, we can allow statements of the form $a \approx_I b$ in specifications, and replace them with a number of write definitions according to the size of I, with fresh element and array variables in every write.

Assuming v_1, we obtain the following valuations for variables $\bar{x}_E$:

$$e_2 := b_k := b_{i-2} := a_i, e_1 := b_i := a'_i := a_{i-2} + 1, a'_k := a'_{i-2} := a_k.$$

Assuming v_2, valuations are the same except for $b_{i-2} := a'_{i-2} := a_{i-2}$. The precondition is in both cases $P_E \equiv a_i > 0 \wedge a_k \leq 0$ (plus the $\mathcal{T}_I$-part of ϕ'').

$\mathcal{T}_I$-Synthesis We obtain $j := \lfloor \frac{i}{2} \rfloor$ and an iterator $\bar{T}_I^*$ of solutions for (i, k):

$T_I^* :=$ (0,2)
T_I^*.next = **let** (k,i) = T_I^* **in**
if(k+1<i) (k+1,i)
else if(i+2<a_u) (0,i+2)
else return UNSAT

along with a precondition $P \equiv a_i > 0 \wedge a_k \leq 0 \wedge a_u > 2$.

Array synthesis: Lifting the witness terms for elements to array b, we obtain

$$\begin{array}{l} b := \mathsf{if}(i-2=k) \\ \quad \mathsf{write}(\mathsf{write}(a, i, a[T_{i-2}]+1), T_k, a[T_i]) \\ \quad \mathsf{else}\ \mathsf{write}(\mathsf{write}(\mathsf{write}(a, i, a[T_{i-2}]+1), T_{i-2}, a[T_i]), T_k, a[T_i]) \end{array}$$

Result: Finally, we obtain the precondition

$$a_u > 2 \wedge \exists n.\ \left((i,k) = T_I^*.\mathsf{next}^{(n)} \ \rightarrow\ a[i] > 0 \wedge a[k] \leq 0\right)$$

and the program[4] in Fig. 1 for computing i, j, k and b from a and a_u.

5.5 Example: Inverting Program Fragments

The synthesis procedure for arrays can also be used to invert given code fragments, e.g. for automatically obtaining a program that reverts (some of) the changes a given piece of code did to some data. Consider the code fragment

```
if(a[i]==0)
        a[i]:=a[i+1]
else if (a[i]==1)
        a[i]:=a[0]
else if (a[i]>1)
        a[i]:=a[i]−1
else a[i]:= a[i]+2
```

which translates into the specification

$$\begin{array}{r} (a_0[i] = 0 \wedge a_1 = \mathsf{write}(a_0, i, a_0[i+1])) \\ \vee\ (a_0[i] = 1 \wedge a_1 = \mathsf{write}(a_0, i, a_0[0])) \\ \vee\ (a_0[i] > 1 \wedge a_1 = \mathsf{write}(a_0, i, a_0[i]-1)) \\ \vee\ (a_0[i] < 0 \wedge a_1 = \mathsf{write}(a_0, i, a_0[i]+2)), \end{array}$$

[4] The code can be significantly simplified by merging parts that are not affected by case distinctions.

```
if(a_u > 2) {
        (i,k) := T_I^* in
        while (¬(a[i] > 0 && a[k] ≤ 0))
                if (T_I^*.hasNext)
                        (i,k) := T_I^*.next
                else throw new Exception("Unsatisfiable constraint.")
        let j = i / 2 in
        if (i−2 = k) {
                bi := a[i−2]+1
                bk := a[i]
        } else {
                bi := a[i−2]+1
                bk := a[i]
                bi2 := a[i−2]
        }
        if (i−2 = k)
            b := write(write(write(a,i,bi),k,bk))
        else
            b := write(write(write(a,i,bi),k,bk),i−2,bi2)
        (b,i,j,k)
} else throw new Exception("Unsatisfiable constraint.")
```

Fig. 1. Example of code generated by array synthesis procedure

where a_0 refers to the pre-, and a_1 to the post-state value of array a. For synthesizing the inverted code, we assume that a_1 is the input, and a_0 the output. The synthesis procedure will return a piece of code

```
if(a[i]==a[i+1])
        a[i]:=0
else if(a[i]=a[0])
        a[i]:=1
else if(a[i]>0)
        a[i]:=a[i]+1
else a[i]:=a[i]−2
```

Since the relation given by the input code does not model a bijection, applying the inverted code after the input code will not result in exactly the same state. However, for a deterministic code, the resulting state will be equivalent with respect to the original piece of code: if we run the original program for the second time from such state, we will get the same final result as when running the program once.

6 Related Work

Term algebras admit quantifier elimination [5,21] and are thus natural candidates for synthesis. Our synthesis procedure is similar to quantifier elimination

when it comes to eliminating variables that are constrained by an equality, with the additional requirement that the witness term be stored to serve in the program. However, we simplified the treatment of disequalities: existing elimination procedures typically rewrite a disequality between a variable and a term into a disjunction of equalities between the same variable and terms constructed with different constructors [5, p.63sqq]. This has the advantage that the language of formulas needs not be extended, allowing for nested quantifiers to be eliminated one after the other. In our synthesis setting, this is not necessary: we can rely on additional computable functions, as we have illustrated with the use of Δ, greatly simplifying the resulting program. A related area of research is compilation of unification in Prolog [1]. This process typically does not require handling of disequalities, so it deals with a simpler language.

Pattern-matching compilation is a task for which specialized procedures for term algebras have been developed [13,23]. When viewed through the prism of synthesis procedures, these algorithms can be thought of as procedures that are specialized for disjunctions of term equalities, and where the emphasis is put on code reuse. We expect that using a combination of our synthesis procedures and common subexpression elimination techniques, one should be able to derive pattern-matching compilation schemes that would support, e.g., disjunctive patterns, non-linear patterns, and could take into account guards referring to integer predicates.

Our synthesis procedure for arrays is based on a reduction of constraints over arrays to constraints in the combined theory of indices and elements. In particular, our reduction is very close to the decision procedure for extensional arrays introduced by Stump et al. [20]. Combination of strongly disjoint theories is also used in the array decision procedure of de Moura and Bjørner [14], but the main focus of their work was to make array decision procedures more efficient by restricted application of fine-grained reduction rules. In the presence of unknown inputs, these techniques are not applicable in general.

Specialization of decision procedures for the purpose of predicate abstraction was considered in [10]. In addition to covering a different set of theories, our results are broader because our process generates not only a satisfiability check but also the values of variables.

7 Conclusions

We presented synthesis procedures for two important theories: algebraic data types and arrays, formulated in a unified framework. Our contribution fills an unexplored area between two approaches: running SMT solvers at run time [7] and using quantifier-elimination-like algorithms to compile specifications. In this paper we have shown that for two important theories supported by SMT solvers, compilation of constraints is also feasible. Moreover, much like SMT can be built and proved correct modularly, synthesis procedures can be combined as well using our framework.

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991) (available online as PDF)
2. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning. Elsevier (2001)
3. Flener, P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers (1995)
4. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for regular specifications over unbounded domains. In: FMCAD, pp. 101–109 (2010)
5. Hodges, W.: A Shorter Model Theory. Cambridge University Press (1997)
6. Jobstmann, B., Bloem, R.: Optimizations for ltl synthesis. In: FMCAD, pp. 117–124 (2006)
7. Köksal, A.S., Kuncak, V., Suter, P.: Constraints as control. In: POPL, pp. 151–164 (2012)
8. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. CACM 55(2), 103–111 (2012)
9. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI, pp. 316–329 (2010)
10. Lahiri, S.K., Ball, T., Cook, B.: Predicate Abstraction via Symbolic Decision Procedures. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 24–38. Springer, Heidelberg (2005)
11. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. CACM 14(3), 151–165 (1971)
12. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. TOPLAS 2(1), 90–121 (1980)
13. Maranget, L.: Compiling pattern matching to good decision trees. In: ML, pp. 35–46 (2008)
14. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD, pp. 45–52 (2009)
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. TOPLAS 1(2), 245–257 (1979)
16. Smith, D.R.: KIDS: A semiautomatic program development system. TSE 16(9), 1024–1043 (1990)
17. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415 (2006)
18. Spielmann, A., Kuncak, V.: Synthesis for unbounded bit-vector arithmetic. In: IJCAR, pp. 499–513 (2012)
19. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010)
20. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS, pp. 29–37 (2001)
21. Sturm, T., Weispfenning, V.: Quantifier elimination in term algebras: The case of finite languages. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) Computer Algebra in Scientific Computing (CASC). TUM München (2002)
22. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)
23. Wadler, P.: The Implementation of Functional Programming Languages: Efficient Compilation of Pattern-matching, ch. 5, pp. 78–103 (1987)

Towards Efficient Parameterized Synthesis*

Ayrat Khalimov, Swen Jacobs, and Roderick Bloem

Graz University of Technology, Austria

Abstract. Parameterized synthesis was recently proposed as a way to circumvent the poor scalability of current synthesis tools. The method uses cut-off results in token rings to reduce the problem to bounded distributed synthesis, and thus ultimately to a sequence of SMT problems. This solves the problem of scalability in the size of the architecture, but experiments show that the size of the specification is still a major issue. In this paper we propose several optimizations of the approach. First, we tailor the SMT encoding to systems with isomorphic processes and token-ring architecture. Second, we extend the cut-off results for token rings and refine the reduction, using modularity and abstraction techniques. Some of our optimizations also apply to isomorphic or distributed synthesis in arbitrary architectures. To evaluate these optimizations, we developed the first completely automatic implementation of parameterized synthesis. Experiments show a speed-up of several orders of magnitude, compared to the original method.

1 Introduction

By automatically generating correct implementations from a temporal logic specification, reactive synthesis tools relieve system developers from manual low-level implementation and debugging. However, existing tools are not very scalable. For instance, Bloem et al. [3] describe the synthesis of an arbiter for the ARM AMBA Advanced High Performance Bus. The results, obtained using RATSY [2], show that both the size of the implementation and the time for synthesis increase steeply with the number of clients that the arbiter can handle. Since an arbiter for $n+1$ clients is very similar to an arbiter for n clients, this is unexpected.

Similar to the AMBA arbiter, many other specifications in verification and synthesis are naturally parameterized in the number of parallel interacting components [15,14]. To address the poor scalability of reactive synthesis tools in the number of components, Jacobs and Bloem [13] introduced a parameterized synthesis approach. A simple example of a parameterized specification is the following LTL specification of a simple arbiter:

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i) \end{array}$$

* This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) through the national research network RiSE (S11406).

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 108–127, 2013.

In parameterized synthesis, we synthesize a building block that can be cloned to form a system that satisfies such a specification, for any number of components.

Jacobs and Bloem [13] showed that parameterized synthesis is undecidable in general, but semi-decision procedures can be found for classes of systems with cut-offs, i.e., where parameterized verification can be reduced to verification of a system with a bounded number of components. They presented a semi-decision procedure for token-ring networks, building on results by Emerson and Namjoshi [8], which show that for the verification of parameterized token rings, a cut-off of 5 is sufficient for a certain class of specifications. Following these results, parameterized synthesis reduces to distributed synthesis in token rings of (up to) 5 identical processes. To solve the resulting problem, a modification of the SMT encoding of the distributed bounded synthesis problem [12] was used.

Experiments with the parameterized synthesis method [13] revealed that only very small specifications could be handled with this encoding. For example, the simple arbiter presented in the beginning can be synthesized in a few seconds for a ring of size 4, which is the sufficient cut-off for this specification. However, synthesis does not terminate within 2 hours for a specification that also excludes spurious grants, in a ring of the same size. Furthermore, the previously proposed method uses cut-off results of Emerson and Namjoshi [8] and therefore inherits a restricted language support and cannot handle specifications in assume-guarantee style [3].

Contributions. We propose several optimizations for the parameterized synthesis method, and extend the supported specification language. Some of the optimizations apply to the bounded synthesis approach in general, others only to special cases like isomorphic synthesis, or synthesis in token rings. In particular, we introduce two kinds of optimizations:

1. We revisit the previously proposed SMT encoding and tailor it to systems with isomorphic processes and token ring architecture. We use a *bottom-up approach* that models the global system as a product of isomorphic processes, in contrast to the top-down approach used before. Also, we encode particular properties of token rings more efficiently by *restricting the class of solutions* without losing completeness.
2. We consider optimizations that are independent of the SMT encoding. This includes *incremental solving*, i.e. generating isomorphic processes in small rings and testing the result in bigger rings. Furthermore, we use *modular generation of synthesis constraints* when different classes of properties can be encoded in token rings of different sizes. For local properties (that require a ring of size two), we introduce an *abstraction* that replaces the second process with assumptions on its behavior, thus reducing the ring of size two to a single process with additional assumptions on its environment. Finally, we show how to simplify specifications by *strengthening*, e.g. removing liveness assumptions from parts that specify safety properties.

For some of the examples we consider in this paper, these optimizations show a speedup of at least 3 orders of magnitude, which means that we need seconds where we would otherwise need hours.

To allow the reader to assess the ideas behind our optimizations, and their correctness, we present in some detail the background of distributed synthesis in Sect. 2, and parameterized synthesis with the original SMT encoding in Sect. 3. In Sect. 4 we introduce optimizations of the SMT encoding. In Sect. 5, we consider extensions of our language, which allow us to support a broader class of specifications. Sect. 6 introduces more general optimizations that are independent of the encoding, and finally Sect. 7 gives experimental results.

2 Preliminaries

We consider the synthesis problem for distributed systems, with specifications in (fragments of) LTL. Given a system architecture A and a specification φ, we want to find implementations of all system processes in A, such that their composition satisfies φ.

Architectures. An *architecture* A is a tuple (P, env, V, I, O), where P is a finite set of processes, containing the environment process env and system processes $P^- = P \setminus \{env\}$, V is a set of Boolean system variables, $I = \{I_i \subseteq V \mid i \in P^-\}$ assigns a set I_i of Boolean input variables to each system process, and $O = \{O_i \subseteq V \mid i \in P\}$ assigns a set O_i of Boolean output variables to each process, such that $\dot{\bigcup}_{i \in P} O_i = V$. In contrast to output variables, inputs may be shared between processes. Wlog., we use natural numbers to refer to system processes, and assume $P^- = \{1, \ldots, k\}$ for an architecture with k system processes.

Implementations. An *implementation* $\mathcal{T}_i$ of a system process i with inputs I_i and outputs O_i is a labeled transition system (LTS) $\mathcal{T}_i = (T_i, t_i, \delta_i, o_i)$, where T_i is a set of states including the initial state t_i, $\delta_i : T_i \times \mathcal{P}(I_i) \to T_i$ a transition function, and $o_i : T_i \to \mathcal{P}(O_i)$ a labeling function.

The *composition* of a set of implementations $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ is the LTS $\mathcal{T}_A = (T_A, t_0, \delta, o)$, with $T_A = T_1 \times \cdots \times T_k$, initial state $t_0 = (t_1, \ldots, t_k)$, labeling function $o : T_A \to \mathcal{P}(\bigcup_{1 \le i \le k} O_i)$ with $o(t_1, \ldots, t_k) = o_1(t_1) \cup \cdots \cup o_k(t_k)$, and transition function $\delta : T_A \times \mathcal{P}(O_{env}) \to T_A$ with

$$\delta((t_1, \ldots, t_k), e) = (\delta_1(t_1, (o(t_1, \ldots, t_k) \cup e) \cap I_1), \ldots, \delta_k(t_k, (o(t_1, \ldots, t_k) \cup e) \cap I_k)),$$

i.e., every process advances according to its own transition function and input variables, where inputs from other system processes are interpreted according to the labeling of the current state.

Asynchronous Systems. An *asynchronous system* is an LTS such that in every transition, only a subset of the system processes changes their state. This is decided by a *scheduler*, which determines for every transition the processes that are allowed to make a step. We assume that the environment is always scheduled, and the scheduler is a part of the environment. Formally, O_{env} contains additional scheduling variables $s_1, \ldots, s_k$, and $s_i \in I_i$ for every i. We require $\delta_i(t, I) = t$ for any i and set of inputs I with $s_i \notin I$.

Token Rings. We consider a class of architectures called *token rings*, where processes can only communicate by passing a token. At any time only one process

can possess the token, and in a ring of size k, a process i which has the token can pass it to process $i+1 \bmod k$ by raising an output $\mathsf{send}_i \in O_i \cap I_{i+1}$. We assume that token rings are implemented as asynchronous systems, where in every step only one system process may change its state, except for token-passing steps, in which both of the involved processes change their state.

Distributed Synthesis. The *distributed synthesis problem* for an architecture A and specification φ, is to find implementations for the system processes of A, such that the composition of the implementations $\mathcal{T}_1, \ldots, \mathcal{T}_k$ satisfies φ, written $A, (\mathcal{T}_1, \ldots, \mathcal{T}_k) \models \varphi$. Specification φ is *realizable* with respect to an architecture A if such implementations exist. Synthesis and checking realizability of LTL specifications have been shown to be undecidable for architectures in which not all processes have the same information wrt. environment outputs [10].

Bounded Synthesis. The *bounded synthesis problem* for given architecture A, specification φ and a family of bounds $\{b_i \in \mathbb{N} \mid i \in P^-\}$ on the size of system processes, as well as a bound b_A for their composition $\mathcal{T}_A$, is to find implementations $\mathcal{T}_i$ for the system processes such that their composition $\mathcal{T}_A$ satisfies φ, with $|\mathcal{T}_i| \leq b_i$ for all process implementations, and $|\mathcal{T}_A| \leq b_A$.

Automata. A *universal co-Büchi tree automaton (UCT)* is a tuple $\mathcal{U} = (\Sigma, \Upsilon, Q, \rho, \alpha)$, where Σ is a finite output alphabet, Υ is a finite input alphabet, $\rho : Q \times \Sigma \times \Upsilon \to \mathcal{P}(Q)$ is the transition relation, and α is the set of rejecting states. We call $\mathcal{U}$ a *one-letter* UCT if $|\Sigma| = |\Upsilon| = 1$. A one-letter UCT is *accepting* if all its paths visit α only finitely often. The *run graph* of a UCT $\mathcal{U}$ on an implementation $\mathcal{T}$ is a one-letter UCT obtained by taking the usual synchronous product and replacing all labels by an arbitrary one. An implementation is accepted if its run graph is accepting. The language of $\mathcal{U}$ consists of all accepted implementations. In the graph defined by Q and δ, we call a strongly connected component (SCC) *accepting* (*rejecting*) if it does not (does) contain states in α.

3 Parameterized Synthesis

In this section we recapitulate the method for parameterized synthesis introduced by Jacobs and Bloem [13].

Parameterized Architectures and Specifications. Let $\mathcal{A}$ be the set of all architectures. A *parameterized architecture* is a function $\Pi : \mathbb{N} \to \mathcal{A}$. A *parameterized token ring* is a parameterized architecture R with
$R(n) = (P^n, env, V^n, I^n, O^n)$, where

- $P^n = \{env, 1, \ldots, n\}$,
- I^n assigns to each process a set I_i of isomorphic inputs. That is, for some I, I_i consists of the inputs in I subscripted with i. Additionally, I_i contains the token-passing input send_{i-1} from process $i-1 \pmod n$.
- Similarly, O^n assigns isomorphic, indexed sets of outputs to all system processes, with $\mathsf{send}_i \in O_i$, and every output of *env* is indexed with all values from 1 to n.

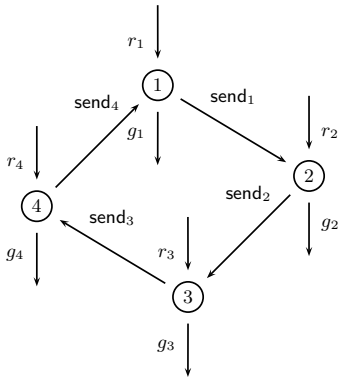


Fig. 1. Token ring with 4 processes

A *parameterized specification* φ is an LTL specification with indexed variables, and universal quantification over indices. We say that a parameterized architecture Π and a process implementation $\mathcal{T}$ *satisfy* a parameterized specification (written $\Pi, \mathcal{T} \models \varphi$) if for any n, $\Pi(n), (\mathcal{T}, \dots, \mathcal{T}) \models \varphi$.

Example 1. Consider the parameterized token ring R_{arb} with $R_{arb}(n) = (P^n, env, V^n, I^n, O^n)$, where

$$\begin{aligned} P^n &= \{env, 1, \dots, n\} \\ V^n &= \{r_1, \dots, r_n, g_1 \dots, g_n, \mathsf{send}_1, \dots, \mathsf{send}_n\} \\ I_i &= \{r_i, \mathsf{send}_{i-1}\} \\ O_{env} &= \{r_1, \dots, r_n\} \\ O_i &= \{g_i, \mathsf{send}_i\} \end{aligned}$$

The architecture $R_{arb}(n)$ defines a token ring with n system processes, with each process i receiving an input r_i from the environment and another input send_{i-1} from the previous process in the ring, and an output send_i to the next process, as well as an output g_i to the environment.

An instance of this parameterized architecture for $n = 4$ is depicted in Fig. 1, and the following is the parameterized specification from the introduction:

$$\begin{aligned} &\forall i \neq j. \;\; \mathsf{G}\,\neg(g_i \wedge g_j) \\ &\forall i. \qquad \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i). \end{aligned}$$

Isomorphic and Parameterized Synthesis. The *isomorphic synthesis problem* for an architecture A and a specification φ is to find an implementation $\mathcal{T}$ for all system processes $(1, \dots, k)$ such that $A, (\mathcal{T}, \dots, \mathcal{T}) \models \varphi$, also written $A, \mathcal{T} \models \varphi$. The *parameterized synthesis problem* for a parameterized architecture Π and a parameterized specification φ is to find an implementation $\mathcal{T}$ for all system processes such that $\Pi, \mathcal{T} \models \varphi$. The *parameterized (isomorphic) realizability problem* is the question whether such an implementation exists.

A *cut-off* for Π and φ is a number $k \in \mathbb{N}$ such that

$$\Pi(k), \mathcal{T} \models \varphi \quad \Rightarrow \quad \Pi(n), \mathcal{T} \models \varphi \text{ for all } n \geq k.$$

3.1 Reduction of Parameterized to Isomorphic Synthesis

Emerson and Namjoshi [8] have shown that verification of LTL\X properties for implementations of parameterized token rings can be reduced to verification of a small ring with up to five processes, depending on the form of the specification.

Theorem 1 ([8]). *Let R be a parameterized token ring, $\mathcal{T}$ an implementation of the isomorphic system processes that ensures fair token passing, and φ a parameterized specification. For a sequence $\overline{t}$ of index variables and terms in arithmetic modulo n, let $f(\overline{t})$ be a formula that only refers to system variables indexed by terms in $\overline{t}$. Then,*

$$R, \mathcal{T} \models \varphi \iff R(k), \mathcal{T} \models \varphi \text{ for } 1 \leq k \leq n,$$

where n is a cut-off depending on the specification: (a) if $\varphi = \forall i.\ f(i)$, then $n = 2$; (b) if $\varphi = \forall i.\ f(i, i+1)$, then $n = 3$, (c) if $\varphi = \forall i \neq j.\ f(i, j)$, then $n = 4$, and (d) if $\varphi = \forall i \neq j.\ f(i, i+1, j)$, then $n = 5$. [1]

Thus, verification of such structures is decidable. For synthesis, we obtain the following corollary:

Corollary 1 ([13]). *For a given parameterized token ring R and parametric specification φ, parameterized synthesis can be reduced to isomorphic synthesis in rings of size 2 (3, 4, 5) for specifications of type a) (b, c, d, resp.).*

Using a modification of undecidability proofs for the distributed synthesis problem [16,10], Jacobs and Bloem [13] showed undecidability of isomorphic synthesis in token rings, which implies undecidability of parameterized synthesis.

3.2 Bounded Isomorphic Synthesis

The reduction from Sect. 3.1 allows us to reduce parameterized synthesis to isomorphic synthesis with a fixed number of processes. To solve the resulting problem, bounded synthesis is adapted for isomorphic synthesis in token rings.

Bounded Synthesis. Following [12], the bounded synthesis procedure consists of three steps:

1. **Automata translation.** The LTL specification φ (including fairness assumptions like fair scheduling) is translated into a UCT $\mathcal{U}$ which accepts an LTS $\mathcal{T}$ iff $\mathcal{T}$ satisfies φ.
2. **SMT Encoding.** Existence of an LTS which satisfies φ is encoded into a set of SMT constraints over the theory of integers and free function symbols. States of the LTS are represented by natural numbers, state labels as free

[1] The results of [8] allow to fix one of the indices in the specification. For $\forall i.\ f(i)$ it is enough to verify the property $f(0)$ under the assumption that initially the token is given to a randomly chosen process. For $\forall i \neq j.\ f(i, j)$ it is enough to verify $\forall j \neq 0.\ f(0, j)$. See Lemma 3 in [8].

functions of type $\mathbb{N} \to \mathbb{B}$, and the global transition function as a free function of type $\mathbb{N} \times \mathbb{B}^{|O_{env}|} \to \mathbb{N}$. To obtain an interpretation of these symbols that satisfies the specification φ, we introduce labels $\lambda_q^{\mathbb{B}} : \mathbb{N} \to \mathbb{B}$ and free functions $\lambda_q^{\#} : \mathbb{N} \to \mathbb{N}$, which are defined such that (i) $\lambda_q^{\mathbb{B}}(t)$ is true iff the product of $\mathcal{T}$ and $\mathcal{U}$ contains a path from an initial state to a state (t, q) with $q \in Q$ and (ii) valuations of the $\lambda_q^{\#}$ must be non-decreasing along paths of $\mathcal{U}$, and strictly increasing for transitions that visit a rejecting state of $\mathcal{U}$. This ensures that an LTS satisfying these constraints cannot have runs which enter rejecting states infinitely often. The corresponding constraint for an UCT $(\Sigma, \Upsilon, Q, \rho, \alpha)$ and an implementation (T, t, δ, o) is

$$\bigwedge_t \bigwedge_I \bigwedge_{q,q'} \lambda_q^{\mathbb{B}}(t) \wedge q' \in \rho(q, o(t), I) \to \lambda_{q'}^{\mathbb{B}}(\delta(t, I)) \wedge \lambda_{q'}^{\#}(\delta(t, I)) \rhd_q \lambda_q^{\#}(t), \quad (1)$$

where $\rhd_q$ equals $>$ if $q \in \alpha$, and $\rhd_q$ equals $\geq$ otherwise. Furthermore, we add a constraint that $\lambda^{\mathbb{B}}$ holds in the initial state of the run graph.
Finally, transition functions of individual processes are defined indirectly by introducing projections $d_i : \mathbb{N} \to \mathbb{N}$, mapping global to local states. To ensure that local transitions of process i only depend on inputs in I_i, we add a constraint

$$\bigwedge_i \bigwedge_{t,t'} \bigwedge_{I,I'} d_i(t) = d_i(t') \wedge I \cap I_i = I' \cap I_i \to d_i(\delta(t, I)) = d_i(\delta(t', I')). \quad (2)$$

3. **Iteration for Increasing Bounds.** To obtain a decidable problem, the number of states in the LTS that we are looking for is bounded, which allows us to instantiate all quantifiers over state variables t, t' explicitly with all values in the given range. If the constraints are unsatisfiable for a given bound, we increase it and try again. If they are satisfiable, we obtain a model, giving us an implementation for the system processes such that φ is satisfied.

Adaption to Token Rings. The bounded synthesis approach is adapted for synthesis in token rings, along with some first optimizations for a better performance of the synthesis method.[2]

- We want to obtain an asynchronous system in which the environment is always scheduled, along with exactly one system process. In general, we could add a constraint $\bigwedge_i \bigwedge_I s_i \notin I \to d_i(\delta(t, I)) = d_i(t)$ (where I is a set of inputs and s_i is the scheduling variable for process i). For our case, we do not need $|P|$ scheduling variables, but can encode the index of the scheduled process into a binary representation with $log_2(|P^-|)$ inputs.
- We use the semantic variation where environment inputs are not stored in system states, but are directly used in the transition term that computes the following state (cp. [12], Sect. 8). This results in an implementation which is a factor of $|O_{env}|$ smaller.

[2] This includes modifications and optimizations mentioned in [12], as well as [13].

– We encode the special features of token rings: i) Exactly one process should have the token at any time; ii) Only a process which has the token can send it; iii) If process i is scheduled, currently has the token, and wants to send it, then in the next state process $i+1$ has the token and process i does not; iv) If process i has the token and does not send it (or is not scheduled), it also has the token in the next state, and v) if process i does not have the token and does not receive it from process $i-1$, then it will also not have the token in the next step. Properties ii) – v) are encoded in the following constraints, where $\mathsf{tok}_i(d_i(t))$ is true in state t iff process i has the token, $\mathsf{send}(d_i(t))$ is true iff i is ready to send the token, and $\mathsf{sched}_i(I)$ is true iff the scheduling variables in I are such that process i is scheduled:

$$\begin{array}{l}
\bigwedge_i \bigwedge_t \bigwedge_I \mathsf{tok}(d_i(t)) \rightarrow (\mathsf{send}(d_i(t)) \wedge \mathsf{sched}_i(I)) \vee \mathsf{tok}(d_i(\delta(t,I))) \\
\bigwedge_i \bigwedge_t \quad \neg\mathsf{tok}(d_i(t)) \rightarrow \neg\mathsf{send}(d_i(t)) \\
\bigwedge_i \bigwedge_t \bigwedge_I \mathsf{send}(d_i(t)) \wedge \mathsf{sched}_i(I) \rightarrow \neg\mathsf{tok}(d_i(\delta(t,I))) \\
\bigwedge_i \bigwedge_t \bigwedge_I \mathsf{send}(d_{i-1}(t)) \wedge \mathsf{sched}_{i-1}(I) \rightarrow \mathsf{tok}(d_i(\delta(t,I))) \\
\bigwedge_i \bigwedge_t \bigwedge_I \neg\mathsf{tok}(d_i(t)) \wedge \neg(\mathsf{send}(d_{i-1}(t)) \wedge \mathsf{sched}_{i-1}(I)) \rightarrow \neg\mathsf{tok}(d_i(\delta(t,I))).
\end{array} \tag{3}$$

We do not encode property i) directly, because it is implied by the remaining constraints whenever we start in a state where only one process has the token.

– Token passing is an exception to the rule that only the scheduled process changes its state: if process i is scheduled in state t, and both $\mathsf{tok}(d_i(t))$ and $\mathsf{send}(d_i(t))$ hold, then in the following transition both processes i and $i+1$ will change their state. The constraint which ensures that only scheduled processes may change their state is modified into

$$\begin{array}{l}
\bigwedge_i \bigwedge_t \bigwedge_I \neg\mathsf{sched}_i(I) \wedge \neg(\mathsf{sched}_{i-1}(I) \wedge \mathsf{tok}(d_{i-1}(t)) \wedge \mathsf{send}(d_{i-1}(t))) \\
\qquad\qquad \rightarrow d_i(\delta(t,I)) = d_i(t).
\end{array} \tag{4}$$

– We use isomorphism constraints to encode that the processes are identical. To this end, we use the same function symbols for state labels of all system processes, and restrict local transitions such that they result in the same local state whenever the previous local states and the visible inputs are equivalent. Since our definition allows processes that are not scheduled to receive the token, we add a rule for this special case. The resulting constraints for local transitions are:

$$\begin{array}{l}
\bigwedge_{i>1} \bigwedge_{t,t'} \bigwedge_{I,I'} d_1(t) = d_i(t') \wedge \mathsf{sched}_1(I) \wedge \mathsf{sched}_i(I') \wedge I \cap I_1 = I' \cap I_i \\
\qquad\qquad \rightarrow d_1(\delta(t,I)) = d_i(\delta(t',I')) \\
\bigwedge_{i>1} \bigwedge_{t,t'} \bigwedge_{I,I'} d_1(t) = d_i(t') \wedge \mathsf{send}(d_n(t)) \wedge \mathsf{send}(d_{i-1}(t')) \\
\qquad\qquad \wedge \mathsf{sched}_n(I) \wedge \mathsf{sched}_{i-1}(I') \wedge I \cap I_1 = I' \cap I_i \\
\qquad\qquad \rightarrow d_1(\delta(t,I)) = d_i(\delta(t',I')).
\end{array} \tag{5}$$

– Finally, a precondition of Thm. 1 is that the implementation needs to ensure fair token-passing. Let $\mathsf{fair\_scheduling}$ stand for $\bigwedge_j \mathsf{G}\,\mathsf{F}\,\mathsf{sched}_j$. Then, we always add

$$\bigwedge_i (\mathsf{fair\_scheduling} \rightarrow (\mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{send}_i))) \tag{6}$$

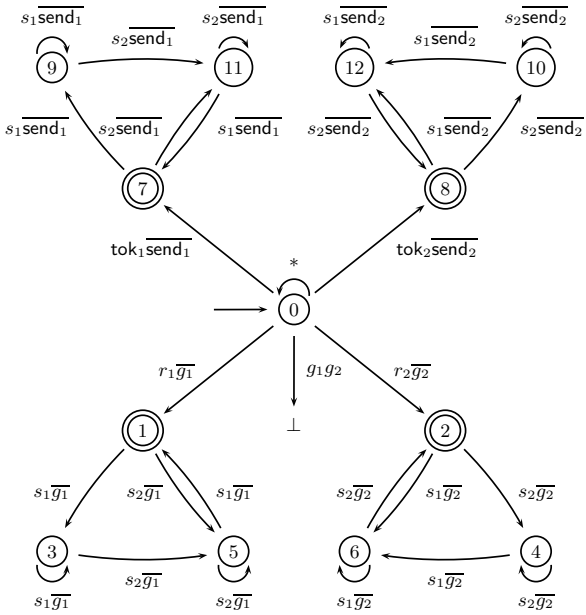


Fig. 2. Universal co-Büchi automaton for Example 2 for two processes

to φ. Similarly, the fair_scheduling assumption needs to be added to any liveness conditions of the specification, as without fair scheduling in general liveness conditions cannot be guaranteed.
Note that these additional formulas need not be taken into account when choosing which case of Thm. 1 needs to be applied.

Example 2. To synthesize an implementation of the simple arbiter from Example 1, we first add constraints for fair scheduling and fair token-passing. The resulting specification is

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{fair\_scheduling} \rightarrow (\mathsf{G}(r_i \rightarrow \mathsf{F}\,g_i)) \\ \forall i. & \mathsf{fair\_scheduling} \rightarrow (\mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{send}_i)). \end{array}$$

For a ring of two processes, this specification translates to the co-Büchi automaton shown in Fig. 2. This automaton is encoded into a set of SMT constraints, part of which is shown in Fig. 3 (only constraints for states $0, 1, 3, 5$ of the automaton are shown). These constraints, together with general constraints for asynchronous systems, isomorphic processes, token rings, and size bounds, are handed to the SMT solver.

Correctness and Completeness of Bounded Synthesis for Token Rings. For a specification φ that is realizable in a token ring of size n, the given semi-algorithm will eventually find an implementation satisfying φ in token rings of

$$
\begin{array}{ll}
 & \lambda_0^{\mathbb{B}}(0) \\
 & \mathsf{tok}(d_1(0)) \wedge \neg \mathsf{tok}(d_2(0)) \\
\forall t.\ \forall I. & \lambda_0^{\mathbb{B}}(t) \rightarrow \lambda_0^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_0^{\#}(\delta(t,I)) \geq \lambda_0^{\#}(t) \\
\forall t. & \lambda_0^{\mathbb{B}}(t) \rightarrow \neg(g(d_i(t)) \wedge g(d_j(t))) \\
\forall t.\ \forall I. & \lambda_0^{\mathbb{B}}(t) \wedge r_1 \in I \rightarrow \lambda_1^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_1^{\#}(\delta(t,I)) > \lambda_0^{\#}(t) \\
\forall t.\ \forall I. & \lambda_1^{\mathbb{B}}(t) \wedge \mathsf{sched}_1(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_3^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_3^{\#}(\delta(t,I)) \geq \lambda_1^{\#}(t) \\
\forall t.\ \forall I. & \lambda_1^{\mathbb{B}}(t) \wedge \mathsf{sched}_2(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_5^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_5^{\#}(\delta(t,I)) \geq \lambda_1^{\#}(t) \\
\forall t.\ \forall I. & \lambda_3^{\mathbb{B}}(t) \wedge \mathsf{sched}_1(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_3^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_3^{\#}(\delta(t,I)) \geq \lambda_3^{\#}(t) \\
\forall t.\ \forall I. & \lambda_5^{\mathbb{B}}(t) \wedge \mathsf{sched}_2(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_5^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_5^{\#}(\delta(t,I)) \geq \lambda_5^{\#}(t) \\
\forall t.\ \forall I. & \lambda_5^{\mathbb{B}}(t) \wedge \mathsf{sched}_1(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_1^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_1^{\#}(\delta(t,I)) > \lambda_5^{\#}(t) \\
\ldots & \ldots
\end{array}
$$

Fig. 3. Constraints for Example 2 for two processes

size n. If φ furthermore falls into one of the classes described in Theorem 1, then the implementation will satisfy φ in token rings of arbitrary size.

4 Optimizations of the Encoding

In this section, we describe a first set of optimizations that make synthesis significantly more efficient. We consider the encoding of the problem into SMT constraints, and aim to remove as much decision freedom from the SMT solver as possible. All optimizations presented in this section are sound and complete. Optimization of *counters* was introduced in [12,7] and applies to bounded synthesis in general. *Bottom-up* encoding is possible in general (to some extent), but will be most useful for isomorphic systems. Finally, *fixed token function* is an optimization specific to token rings (or token-passing systems in general).

Counters. As mentioned in Sect. 3, labeling functions $\lambda^{\#}$ count the visits to rejecting states, and a satisfying valuation for them exists only if all run paths visit rejecting states only finitely often. In a run path, a repeated visit to the same rejecting state is possible only if the path stays in an SCC of the specification UCT $\mathcal{U}$. Therefore, we can reduce the number of $\lambda^{\#}$ annotations by introducing them only for states of $\mathcal{U}$ that are in a rejecting SCC. This optimization reduces the number of counters as well as their maximum value significantly.

Example 3. In the simple arbiter, this optimization means that we do not need $\lambda^{\#}$ annotations in the initial state. The benefit becomes more visible in a full arbiter, which in addition requires that there are no spurious grants and that every grant is lowered eventually. A simplified UCT for such a specification is given in Fig. 4. Besides mutual exclusion of the grants, this presentation of the UCT only shows the constraints for arbiter i. In the figure, s_i stands for sched_i and $a_i = \mathsf{sched}_i \vee \mathsf{send}_{i-1}$ means the process is active and can react to the environment input. Note that the SCCs around states 2 and 3 only reject rings with fair scheduling — they become larger when processes are added. For this UCT, counters $\lambda^{\#}$ are only needed for states 3, 6, 8, 2, 5, and 7.

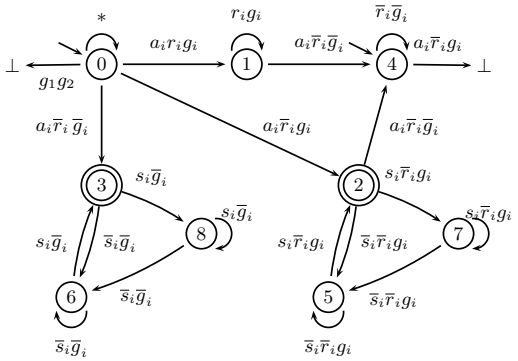


Fig. 4. UCT of the full arbiter

Bottom-Up. In the original approach described in Sect. 3, the SMT solver searches for a global transition function and projection functions d_i that satisfy input dependence, scheduling, and isomorphism constraints (2) (4) (5). Instead, we propose to go bottom-up: to search for a single process transition function and build the global one from local ones. Thus, all the processes share the same transition function symbol – this ensures their isomorphism, and constraints (5) can be removed. Also, process transitions functions now depend only on corresponding to the process inputs, and we can safely remove constraints (2). In addition, we wrap the transition function into an auxiliary function which calls the original if the process is scheduled or receives the token, and otherwise returns the current process state. This obviates the need for constraints (4).

$$
\begin{array}{lll}
 & & \lambda_0^{\mathbb{B}}(0,1) \\
 & & \mathsf{tok}(0) \wedge \ \neg\mathsf{tok}(1) \\
\forall t_1.\ \forall t_2.\ \forall I_1.\ \forall I_2. & & \lambda_0^{\mathbb{B}}(t_1,t_2) \ \rightarrow \ \lambda_0^{\mathbb{B}}(t_1',t_2') \\
\forall t_1.\ \forall t_2. & & \lambda_0^{\mathbb{B}}(t_1,t_2) \ \rightarrow \ \neg(g(t_1) \wedge g(t_2)) \\
\forall t_1.\ \forall t_2.\ \forall I_1.\ \forall I_2. & & \lambda_0^{\mathbb{B}}(t_1,t_2) \wedge r_1 \in I_1 \ \rightarrow \ \lambda_1^{\mathbb{B}}(t_1',t_2') \\
\forall t_1.\ \forall t_2.\ \forall I_1\ .\forall I_2. & & \lambda_1^{\mathbb{B}}(t_1,t_2) \wedge \mathsf{sched}_2 \wedge \neg g(t_1) \ \rightarrow \ \lambda_5^{\mathbb{B}}(t_1',t_2') \wedge \lambda_5^{\#}(t_1',t_2') \geq \lambda_1^{\#}(t_1,t_2) \\
\forall t_1.\ \forall t_2.\ \forall I_1\ .\forall I_2. & & \lambda_5^{\mathbb{B}}(t_1,t_2) \wedge \mathsf{sched}_2 \wedge \neg g(t_1) \ \rightarrow \ \lambda_5^{\mathbb{B}}(t_1',t_2') \wedge \lambda_5^{\#}(t_1',t_2') \geq \lambda_5^{\#}(t_1,t_2) \\
\forall t_1.\ \forall t_2.\ \forall I_1\ .\forall I_2. & & \lambda_5^{\mathbb{B}}(t_1,t_2) \wedge \mathsf{sched}_1 \wedge \neg g(t_1) \ \rightarrow \ \lambda_1^{\mathbb{B}}(t_1',t_2') \wedge \lambda_1^{\#}(t_1',t_2') > \lambda_5^{\#}(t_1,t_2)
\end{array}
$$

Fig. 5. Some of constraints for Example 2 for two processes using the Bottom-up encoding and Counters optimizations; $t_i' = \delta(t_i, I_i)$, δ is a wrapped local transition function; labeling functions changed its type: $\lambda_q^{\#/B} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}/\mathbb{B}$.

Similar approach that uses process transition functions explicitly was proposed in [11], but they still keep projection functions and a global transition function to describe the system. We removed these notions at all (see Fig. 5).

On the downside, this optimization does not allow us to bound the number of global states independently of the number of local states as in the original approach [13] or in [11] (and the number of global states is always equal to $|T|^n$).

Fixed Token Function. In the original approach, possession of the token is encoded by an uninterpreted function tok. We fix process states without/with a token $T_{\neg tok}/T_{tok}$ and define $\mathsf{tok}(t) := (t \in T_{tok})$, thus exempting the SMT solver from finding a valuation for tok. Fixing token possession functions has two important consequences.

First, it allows us to precompute global states with exactly one token in a ring. In case of 3 processes, global states are $\{(t_*, t, t) \cup (t, t_*, t) \cup (t, t, t_*)\}$, where $t_* \in T_{tok}, t \in T_{\neg tok}$. Then we build main constraints (1) only for these precomputed global states, and ignore other, invalid, global states. The system cannot move into an invalid global state with number of tokens different from one due to token ring constraints (3). In the original approach, invalid global states constitutes most of the global state space, and ignoring them reduces the state space significantly (exponentially in number of processes), leading to smaller SMT queries.

The second consequence of fixing tok is a possible restriction of generality of solutions. Different separations $T_{\neg tok}/T_{tok}$ may lead to different solutions. In general, systems with larger ratio $p = |T_{\neg tok}|/|T_{tok}|$ have a larger global state space, and processes without a token have more possibilities for transitions. With a maximal p, the system is completely parallel, and a minimal p leads to a sequential processing. The choice of p also affects the synthesis time because it changes the number of global states and, therefore, the size of the query.

A related optimization is to use binary encoding for token possession and sending that would automatically remove bad states from the consideration.

5 Extensions of Supported Language

Before introducing a second set of optimizations, we consider some extensions of our specification language, enabling us to treat more interesting examples. While the results of Emerson and Namjoshi [8] give us cut-offs that allow for parameterized synthesis in principle, a closer inspection of examples from the literature shows that the supported language is not expressive enough to handle many of them. Consider the parameterized arbiter specification introduced by Piterman, Pnueli and Sa'ar [15] with a specification of the form

$$\mathsf{Assume} \rightarrow \mathsf{Guarantee}, \text{ where}$$

$$\mathsf{Assume} \equiv \bigwedge_i \left(\overline{r_i} \wedge \mathsf{G}\left((r_i \neq g_i) \rightarrow (r_i = \mathsf{X}\, r_i)\right) \wedge \mathsf{G}\left((r_i \wedge g_i) \rightarrow \mathsf{F}\,\overline{r_i}\right)\right)$$

$$\mathsf{Guarantee} \equiv \bigwedge_{i \neq j} \left(\mathsf{G}\,\neg(g_i \wedge g_j)\right) \wedge \bigwedge_i \left(\overline{g_i} \wedge \begin{pmatrix} \mathsf{G}((r_i = g_i) \rightarrow (g_i = \mathsf{X}\, g_i)) \\ \wedge\ \mathsf{G}((r_i \wedge \overline{g_i}) \rightarrow \mathsf{F}\, g_i) \\ \wedge\ \mathsf{G}((\overline{r_i} \wedge g_i) \rightarrow \mathsf{F}\,\overline{g_i}) \end{pmatrix}\right).$$

This specification points to three limitations in the language considered by Emerson and Namjoshi:

First, the conjunction over all processes in the assumption turns into a disjunction when we bring the formula into a prenex form. Disjunctions over processes are however not supported by Emerson and Namjoshi. Second, this formula will quantify over (at least) three independent variables, which is also not supported in their framework. We will show that these two limitations can be effectively overcome in token-rings by using more general results by Clarke et al. [5] on network decomposition.

Finally, the specifications of Assume and Guarantee contain the X operator, which is completely excluded from the language of both Emerson/Namjoshi and Clarke et al. One may assume that one of the reasons for excluding the X operator is that, in asynchronous distributed systems, the presence of a (fair but otherwise arbitrary) scheduler can invalidate statements about the next state of a given process by simply not scheduling it. We will make some observations about cases where the usual X operator can still be used, and furthermore introduce a local variant of the X operator, that takes scheduling of a process into account.

5.1 Network Decomposition for Token Rings

The results by Clarke et al. allow for specifications with both conjunctions and disjunctions over all processes, and also allow an arbitrary number of index variables. For checking a property ϕ in a token-passing network, the consider decompositions into possible network topologies, where processes that are not represented by an index in ϕ are replaced by so-called *hubs* which simply pass on the token. A *k-indexed property* is a formula with arbitrary quantification that refers to system variables of at most k different processes. Their main result is

Theorem 2 ([5]). *For checking any k-indexed LTL\X property ϕ in token-passing networks, it is sufficient to check ϕ on up to $3^{k(k-1)}2^k$ network topologies of size up to $2k$.*

However, in general this result is not constructive, a suitable decomposition into network topologies still needs to be found (for arbitrary network architectures). For the case of token rings, it is easy to find the suitable decomposition: for every number of processes n, there is only one ring of this size. Thus, Thm. 2 directly implies that for any k-indexed property, it is sufficient to check it on rings of all sizes up to $2k$. We can even get a result that is a bit stronger (in that it does not require to check small rings if we only consider rings of size $> 2k$):

Corollary 2. *If ϕ is a k-indexed property and Π a parameterized token-ring architecture, then $2k$ is a cut-off for ϕ.*

Proof. Suppose ϕ holds in a ring of size $2k$, but not in some bigger ring. This means that there is a tuple $(p_1, \ldots, p_k)$, or a set of such tuples, such that for this combination of processes, ϕ is not satisfied. Using the terminology of Clarke et al., each of these tuples defines a network topology, by abstracting processes that are not in the tuple into hubs, where all neighboring processes are abstracted into one hub. By Clarke et al., every network with the same topology will not satisfy

ϕ. Since the size of this network topology is at most $2k$ (hubs and processes p_i alternating), we can find tuples of processes with the same topology in the ring of size $2k$. Thus, ϕ cannot hold in the ring of size $2k$. Contradiction. □

With this result, parameterized synthesis in token rings can be extended to include specifications with an arbitrary number of quantified variables, and arbitrary quantifier alternations.

5.2 Handling the X Operator

In the example given above, all X operators are used in a way that forbids change as long as some conditions hold. We note that this special usage of the X operator is not a problem when we use this specification in asynchronous distributed systems, for two reasons:

1. In this use case, X operators do not make the specification unrealizable because of the scheduling. Indeed, the environment cannot simply invalidate the property by not scheduling the process, since it will trivially hold in the next step if the process controlling the output does not change it.
2. Cut-off results we use holds for LTL\X, but they still hold for this specification, since we can always rewrite such usage of X operators into a form without X: $\mathsf{G}(\varphi \rightarrow p = \mathsf{X}\, p) \Leftrightarrow \mathsf{G}(\varphi \rightarrow p\, \mathsf{W}\, \neg\varphi) \wedge \mathsf{G}(\varphi \rightarrow \neg p\, \mathsf{W}\, \neg\varphi)$.

In addition to this special usage of the usual X operator, we can consider examples with a local variant X_i useful for specifying Globally Asynchronous Locally Synchronous [4] systems. The local X_i specifies the next state from the perspective of process i,

$$\mathsf{X}_i\, p_i \Leftrightarrow \mathsf{F}(\mathsf{sched}_i \wedge \mathsf{X}\, p_i).$$

Obviously, this operator is insensitive to scheduling (in the sense that the environment cannot invalidate properties by not scheduling the process). Furthermore, all our cut-off results still hold for specifications that are conjunctions $\varphi_1 \wedge \ldots \wedge \varphi_n$, if we allow X_i to appear only in local properties φ_i.

6 General Optimizations

In this section we describe high-level optimizations that are not specific to the SMT encoding. The first two optimizations, *incremental solving* and *modular generation of constraints*, are sound and complete. The third, *specification strengthening*, is based on automatic rewriting of the specification and introduces incompleteness. The last optimization, *hub abstraction* is sound and complete.

Modular generation of constraints and specification strengthening apply to bounded synthesis (although the first is particularly useful for parameterized synthesis), while incremental solving only applies to parameterized synthesis. Hub abstraction is specific to token-passing systems.

Incremental Solving. Corollary 2 states that it is sufficient to synthesize a token ring of size $2k$ for k-indexed properties. However, a solution for a smaller number of processes can still be correct in bigger rings. We propose to proceed incrementally, synthesizing first a ring of size 1, then 2, etc., up to $2k$. After synthesizing a process that works in a ring of size n, we check whether it satisfies the specification also in a ring of size $n+1$. Only if the result is negative, we start the computationally much harder task to synthesize a ring of size $n+1$.

Modular Generation of Constraints for Conjunctive Properties. A very useful property of the SMT encoding for parameterized synthesis is that we can separate conjunctive specifications into their parts, generate constraints for the parts separately, and finally search for a solution that satisfies the conjunction of all constraints. In the following, for a parametric specification φ and a number of processes k, let $C(\varphi, k)$ be the set of SMT constraints generated by the bounded synthesis procedure (for a fixed parameterized architecture Π).

We start from the following observation: let $\varphi_1 \wedge \varphi_2$ be a parametric specification, Π a parameterized architecture, $\mathcal{T}$ a process implementation. If $\Pi, \mathcal{T} \models \varphi_1$ and $\Pi, \mathcal{T} \models \varphi_2$, then $\Pi, \mathcal{T} \models \varphi_1 \wedge \varphi_2$. While this may seem trivial, when combined with Thm. 1 and Cor. 1, we obtain the following[3]

Theorem 3. *Let Π be a parameterized architecture and $\varphi_1 \wedge \varphi_2$ a parametric specification, s.t. n_1 is a cut-off for φ_1, and n_2 a cut-off for φ_2 in Π. Then,*

$$\mathcal{T} \models C(\varphi_1, n_1) \wedge C(\varphi_2, n_2) \quad \Rightarrow \quad \Pi(k), \mathcal{T} \models \varphi_1 \wedge \varphi_2 \text{ for } k \geq max(n_1, n_2).$$

In parameterized synthesis, this not only allows us to use separate sets of constraints to ensure different parts of the specification, but also to use different cut-offs for different parts. By conjoining the resulting constraints of all parts, we obtain an SMT problem s.t. any solution will satisfy the complete specification. For a specification like

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i), \end{array}$$

this allows us to separate the global safety condition from the local liveness condition. Then, only for the former we need to generate constraints for a ring of size 4, while for the latter a ring of size 2 is sufficient. This is particularly useful for specifications where the local part is significantly more complex than the global part, like our more complex arbiter examples.

Specification Strengthening. To simplify the specification in assume-guarantee style, we remove some of its assumptions with two rewriting steps. These steps are sound but incomplete, and lead to more robust specifications.

Consider a specification in assume-guarantee style $A_L \wedge A_S \rightarrow G_L \wedge G_S$ with liveness and safety assumptions and guarantees. Our first strengthening is based on the intuition that in practice A_L is not needed to obtain G_S, so we strengthen

[3] Note that, in a slight abuse of notation, we use $\mathcal{T}$ both for the model of the SMT constraints, and for the implementation represented by this model.

the formula to $(A_S \rightarrow G_S) \wedge (A_L \wedge A_S \rightarrow G_L)$. This step is incomplete for specifications where the system can falsify liveness assumptions A_L and therefore ignore guarantees, or if the assumptions $A_S \wedge A_L$ are unrealizable but A_S is realizable. We assume that such cases are not important in practice[4].

Our second strengthening "localizes" assumptions and guarantees. Consider a 2-indexed specification in assume-guarantee style: $\bigwedge_i A_i \rightarrow \bigwedge_j G_j$. Adding assumptions on fairness of scheduling and token ring guarantees, we get:

$$\bigwedge_i \mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \rightarrow \bigwedge_j G_j$$
$$\bigwedge_i \mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \rightarrow \bigwedge_j TR_j$$

where TR_j are token ring constraints (3), A_i, G_j are assumptions and guarantees referring only to a single process[5]. The truth of $\wedge_j TR_j$ implies the truth of fair token passing $\wedge_j \mathsf{G}\,\mathsf{F}\,\mathsf{tok}_j$, therefore we can add it as an assumption to the first line (logically equivalent to the original specification). After that we "localize" the specification by letting every implication only refer to one process (sound, but incomplete) and get the final strengthened specification:

$$\bigwedge_i (\mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \wedge \mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i \rightarrow G_i)$$
$$\bigwedge_i (\mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \rightarrow TR_i)$$

The second step, where we add $\wedge_i \mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ as an assumption to the first constraint, is crucial. Otherwise, the final specification becomes too restrictive and we may miss solutions. The reason why $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ may prevent this is that $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ may work as a local trigger of a violation of an assumption. This is confirmed in the "Pnueli" arbiter experiment, where a violation of one of the assumptions A_i prevents fair token passing in the ring, falsifying $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_j$ for all $j \neq i$.

Filiot et al. in [9] proposed a rewriting heuristic which is essentially a localization step mentioned above. Our version is slightly different since we add $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ assumptions before localization to prevent missing the solutions.

These two strengthenings may change the type, and hence the cut-off, of a specification. For example, after strengthening, the "Pnueli" arbiter specification changes its type from 3-indexed to 2-indexed. Furthermore, most properties become local, and can be efficiently synthesized with the following optimization.

Hub-Abstraction. From Clarke et al. [5] it follows that checking local properties in a token ring is equivalent to checking properties of one process in a ring of size 2, while the second process is a *hub* process which is only required to pass tokens it receives. Instead of introducing an explicit hub process, we model its behavior with environment assumptions A_{hub} for the first process: i) if the process does not have the token, then the environment will finally send the token, ii) if the process has the token, then the environment can not send the token:

$$\mathsf{G}(\neg\mathsf{tok} \rightarrow \mathsf{F}\,\mathsf{send}_{hub})$$
$$\mathsf{G}(\mathsf{tok} \rightarrow \neg\,\mathsf{send}_{hub}).$$

[4] For example, well known class of GR1 specifications [3] used to describe some industrial systems does not use liveness assumptions for safety guarantees. Specifications with unrealizable assumptions likely contain designer errors.

[5] This can be generalized to k-indexed assumptions and guarantees.

We add the hub assumptions A_{hub} above to the original specification and synthesize a single process. Therefore the final property to synthesize becomes:

$$\mathsf{G\,F\,sched} \wedge A \wedge A_{hub} \rightarrow G \wedge TR.$$

This property is equivalent to the original one, that is, the abstraction step is sound and complete. The abstracted property is more complex than the original one, but it can be synthesized in a single process setting. Therefore we trade size of a token ring to be synthesized for the size of the specification.

We can go further and replace $\mathsf{G\,F\,sched}$ with *true*, which can introduce unsoundness in general. But this step is still sound for the class of specifications mentioned in Sect. 5.2, where the environment cannot violate guarantees by not scheduling the process. This is true for all examples we consider in this paper, and a reasonable assumption for many asynchronous systems.

7 Experiments

For the evaluation of optimizations we developed an automatic parameterized synthesis tool that 1) identifies the cut-off of a given LTL specification 2) adds token ring constraints and fair scheduling assumptions to the specification 3) translates the modified specification into a UCT using LTL3BA [1] 4) for a given cut-off and model size bound encodes the automaton into SMT constraints 5) solves the constraints using SMT solver Z3 v.4.1 [6]. If the solver reports unsatisfiability, then no model for the current bound exists, and the tool goes to step 4 and increases the bound until the user interrupts execution or a model is found. A model synthesized represents a Moore machine that can be cloned and stacked together into a token ring of any size.

We have run our experiments on a single core of a Linux machine with two 4-core 2.66 GHz Intel Xeon processors and 64 GB RAM. Reported times in tables include all the steps of the tool. For long running examples, SMT solving contributes most of the time.

For the evaluation of optimizations we run the tool, with different sets of optimizations enabled, on three examples: a simple arbiter, a full arbiter, and a "Pnueli" arbiter. We show solving times in seconds in Table 1 and Table 2. The horizontal axis of the table has columns for token rings of different sizes – up to a cut-off size – 4 for simple and full arbiters, and 6 for "Pnueli" arbiter.

7.1 Encoding Optimizations

Each successive optimization below includes previous optimizations as well.

Original. The implementation of the original version is described in Sect. 3. It starts with a global transition function and uses projection functions and SMT constraints to specify the underlying architecture and isomorphism of processes.

Counters. We use SCC-based counters for rejecting states, minimizing the necessary annotations of our implementations.

Table 1. Effect of encoding optimizations on synthesis time (in seconds, t/o=2h)

	simple4	full2	full3	full4	pnueli2	pnueli3	pnueli4	pnueli5/6
original	11	t/o	t/o	t/o	52	t/o	t/o	t/o
counters	8	2316	t/o	t/o	19	t/o	t/o	t/o
bottom-up	3	24	934	t/o	23	6737	t/o	t/o
fixed tok function	1	2	28	327	7	252	5691	t/o
total speedup	11	$\geq 10^3$	$\geq 10^2$	≥ 20	7	≥ 30	≥ 1.5	-

Bottom-Up. In this version we use the same local transition and output symbols for all the processes. The significant speedup (two orders of magnitude for full2) is caused by two factors: the number of unknowns gets smaller (we don't need projection functions) and the size of SMT query becomes smaller (no need for constraints (2),(4),(5)).

Fixed Token Function. In this version SMT queries contain constraints only for global states with exactly one token in a ring. It is possible because we hard-code tok by dividing a process state space into two equal sets of states without/with a token (1/2 in case of 3 states). Similar to the previous, this optimization is efficient because it reduces the size of SMT queries and the number of unknowns.

In all experiments, constraints were encoded in AUFLIA logic. We also tried bitvector and real number encodings, but with no considerable speed-up.

7.2 General Optimizations

Each successive optimization below includes previous optimizations except Fixed token function, since it is not clear how to divide process state space in a general way. As a "non-optimized" reference version we use bottom-up implementation.

Incremental Solving. Solving times can be sped up considerably by synthesizing a ring of size 2, and checking whether the solution is correct for a ring of size 4. For instance, for the full arbiter, the general solution is found in 24 seconds when synthesizing a ring of size 2 (time from the "bottom-up" row in Table 1). Checking that the solution is correct for a ring of size 4 takes additional 30 seconds, thus reducing the synthesis time from more than 2 hours to 54 seconds. Times for incremental solving are not given in the table.

Strengthening. This version refers to two optimizations described in Sect. 6 - localizing of assume-guarantee properties and removing liveness assumptions from properties with safety guarantees. Specification rewriting works very well, significantly reducing the size of the specification automaton: for example, the automaton corresponding to the "Pnueli" arbiter in a ring of size 4 after rewriting reduces its size from 1700 to 31 nodes (from 41 to 16 for the full arbiter). Also, this optimization changes the cut-off from 6 to 4 for the "Pnueli" arbiter. We left token rings of size 5/6 in rows below to demonstrate scalability of optimizations.

Table 2. Effect of general optimizations on synthesis time (in seconds, t/o=2h)

	simple4	full2	full3	full4	pnueli2	pnueli3	pnueli4	pnueli5	pnueli6
bottom-up	3	24	934	t/o	23	6737	t/o	t/o	t/o
strengthening	1	6	81	638	2	13	90	620	6375
modular	1	4	8	13	2	4	11	49	262
async hub	1	2	2	5	2	3	9	37	236
sync hub	1	1	2	4	2	3	8	42	191
total speedup	3	20	10^2	$\geq 10^3$	10	10^3	$\geq 10^3$	$\geq 10^2$	≥ 40

Modular. In this version, constraints for specifications of the form $\phi_i \wedge \phi_{i,j}$ are generated separately for local properties ϕ_i and for global properties $\phi_{i,j}$, using the same symbols for transition and output functions. Constraints for ϕ_i are generated for a ring of size 2, and constraints for $\phi_{i,j}$ for a ring of size 4. These sets of constraints are then conjoined in one query and fed to the SMT solver. Such separate generation of constraints leads to smaller automata and queries, resulting in approximately 10x speed up.

Hub Abstractions. By replacing one of the processes in a ring of size 2 with assumptions on its behavior, we reduce the synthesis of a ring of size two to the synthesis of a single process. In row "async hub" the process is synthesized in an asynchronous setting, while in row "sync hub" the process is assumed to be always scheduled. The results do not show a considerable speed up, but this optimization might work in cases of larger specifications.

Remarks. It should be noted that our set of experiments is relatively small, and that SMT solvers are sensitive to small changes in the input. Thus, the experiments would certainly benefit from a larger set of benchmarks, and the individual comparison of any two numbers in the table should be taken with a grain of salt. At the same time, the table shows a clear and significant improvement of the solving time when all optimizations are turned on.

8 Conclusions

We showed how optimizations of the SMT encoding, along with modular application of cut-off results, strengthening and abstraction techniques, leads to a significant speed-up of parameterized synthesis. Experimental results show speed-ups of more than three orders of magnitude for some examples. We also showed that using the X operator does not necessarily break cut-off results or make the specification unrealizable in an asynchronous setting. Finally, we applied cut-off results from verification of general token passing systems [5] to synthesis in token rings, thus extending the specification language that the parameterized synthesis method [13] can handle.

The current bottleneck of SMT-based bounded (and thus, parameterized) synthesis is the construction of the UCT automaton. In our experiments, LTL3BA

could not generate the UCT for an AMBA arbiter with only 1 client within two hours. Therefore, we think that it will be important to develop techniques that help us to avoid construction of the whole automaton (for example by separate tracking of assumptions and guarantees violations, as in [7]).

Acknowledgments. We thank Helmut Veith for inspiring discussions on parameterized systems, Bernd Finkbeiner and Sven Schewe for discussions on distributed and bounded synthesis, and Leonardo de Moura for help with Z3.

References

1. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi Automata Translation: Fast and More Deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012)
2. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – A New Requirements Analysis Tool with Synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010)
3. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences 78, 911–938 (2012)
4. Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. Ph.D. thesis, Stanford Univ., CA (1984)
5. Clarke, E.M., Talupur, M., Touili, T., Veith, H.: Verification by Network Decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Ehlers, R.: Symbolic bounded synthesis. Formal Methods in System Design 40, 232–262 (2012)
8. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. International Journal of Foundations of Computer Science 14, 527–549 (2003)
9. Filiot, E., Jin, N., Raskin, J.F.: Antichains and compositional algorithms for LTL synthesis. Form. Methods Syst. Des. 39(3), 261–296 (2011)
10. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Logic in Computer Science (LICS), pp. 321–330. IEEE Computer Society Press (2005)
11. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Proc. Workshop on Automated Formal Methods, pp. 69–76. ACM (2007)
12. Finkbeiner, B., Schewe, S.: Bounded synthesis. Int. J. on Software Tools for Technology Transfer, 1–21 (2012)
13. Jacobs, S., Bloem, R.: Parameterized Synthesis. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 362–376. Springer, Heidelberg (2012)
14. Katz, G., Peled, D.: Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 117–132. Springer, Heidelberg (2011)
15. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)
16. Pnueli, A., Rosner, R.: Distributed systems are hard to synthesize. In: Foundations of Computer Science (FOCS), pp. 746–757. IEEE Computer Society Press (1990)

Automatic Inference of Necessary Preconditions

Patrick Cousot[1], Radhia Cousot[2], Manuel Fähndrich[3], and Francesco Logozzo[3]

[1] NYU, ENS, CNRS, INRIA
pcousot@cims.nyu.edu
[2] CNRS, ENS, INRIA
rcousot@ens.fr
[3] Microsoft Research
{maf,logozzo}@microsoft.com

Abstract. We consider the problem of *automatic* precondition inference. We argue that the common notion of *sufficient* precondition inference (*i.e.*, under which precondition is the program correct?) imposes too large a burden on callers, and hence it is unfit for automatic program analysis. Therefore, we define the problem of *necessary* precondition inference (*i.e.*, under which precondition, if violated, will the program *always* be incorrect?). We designed and implemented several new abstract interpretation-based analyses to infer atomic, disjunctive, universally and existentially quantified necessary preconditions.

We experimentally validated the analyses on large scale industrial code. For unannotated code, the inference algorithms find necessary preconditions for almost 64% of methods which contained warnings. In 27% of these cases the inferred preconditions were also *sufficient*, meaning all warnings within the method body disappeared. For annotated code, the inference algorithms find necessary preconditions for over 68% of methods with warnings. In almost 50% of these cases the preconditions were also sufficient. Overall, the precision improvement obtained by precondition inference (counted as the additional number of methods with no warnings) ranged between 9% and 21%.

1 Introduction

Design by Contract [28] is a programming methodology which systematically requires the programmer to provide the preconditions, postconditions and object invariants (collectively called contracts) at design time. Contracts allow automatic generation of documentation, amplify the testing process, and naturally enable assume/guarantee reasoning for divide and conquer static program analysis and verification. In the real world, relatively few methods have contracts that are sufficient to prove the method correct. Typically, the precondition of a method is weaker than necessary, resulting in unproven assertions within the method, but making it easier to prove the precondition at call-sites. Inference has been advocated as the holy grail to solve this problem.

In this paper we focus on the problem of computing *necessary preconditions* which are *inevitable checks* from within the method that are hoisted to the

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 128–148, 2013.

method entry. This should be contrasted to *sufficient preconditions* which guarantee the absence of possible assertion violations inside the method but may rule out good runs. Programmers will object to the inference of too strong preconditions as they introduce more false warnings at call sites.

Contribution. The basis for this work is [11] which introduced the theoretical framework formalizing the notion of necessary precondition inference. We extend this theoretical framework by introducing the inter-method analysis, by refining the intra-method analyses combining them with `Clousot` [16], by identifying under which circumstances the generated precondition is also sufficient, by showing how one can infer existential preconditions, by introducing a simplification step for scalability, by adding the provenance relation, and by implementing and validating the approach on realistic code bases.

2 Semantics

The semantics of a given program `P` (*e.g.* a method) is a non-empty set $\mathcal{S}$ of runs modeled as finite or infinite execution traces over states Σ. $\mathcal{S}$ can be partitioned into disjoint subsets $\mathcal{S} = \mathcal{E} \cup \mathcal{T} \cup \mathcal{I}$ where the traces in $\mathcal{E}$ are finite bad runs terminating in an error state, the traces in $\mathcal{T}$ are finite good runs terminating in a correct state, and the infinite traces in $\mathcal{I}$, which correspond to non-termination. If X is a set of traces and $s \in \Sigma$, we write $X(s)$ for the set of traces in X starting from state s.

Assertions are either induced by the language semantics (*e.g.*, null-pointer dereference, division by zero, array out of bounds, ...) or they are annotations in the source text (programmer-provided assertions, preconditions, and postconditions). Boolean expressions are side effect free and they are always well-defined when evaluated with shortcut semantics for conjunctions and disjunctions. The set $\mathbb{A}$ denotes the set of the potential failures of `P`. $\mathbb{A}$ contains pairs $\langle \mathtt{c}, \mathtt{b} \rangle$, where `b` is an assertion at the program point `c`. In general, there may be more than one assertion per program point. The bad runs $\mathcal{E} \triangleq \{\sigma s' \mid \exists \langle \mathtt{c}, \mathtt{b} \rangle \in \mathbb{A} : \pi s' = \mathtt{c} \wedge \neg [\![\mathtt{b}]\!] s'\}$ are all traces $\sigma s'$ ending in a state $s' \in \Sigma$ at a control point $\pi s' = \mathtt{c}$ where the evaluation of a language or programmer assertion `b` fails, that is $[\![\mathtt{b}]\!] s'$ is `false`.

Given a formula $C \Rightarrow S$, we say that C is a sufficient condition for S and S is a necessary condition for C. A *sufficient condition* for a statement S is a condition that, *if* satisfied, ensures S's correctness. A *necessary condition* for a statement S *must* be satisfied for the statement to be true.

The first step in our precondition inference algorithm is the collection of all failure points $\langle \mathtt{c}, \mathtt{b} \rangle$ from which we will then try to derive *necessary* preconditions. In practice, the candidate assertions in $\mathbb{A}$ are those assertions which cannot be statically proven by `cccheck` [16](or similar tools). We will use the assertions in $\mathbb{A}$ to infer necessary preconditions. This consists in propagating these conditions backwards to the origin of the traces of the semantics $\mathcal{S}$, at the `entry` control point. The inference of termination preconditions is a separate problem [10], so we ignore the non-terminating behaviors $\mathcal{I}$, or equivalently, assume termination *i.e.* $\mathcal{I} = \emptyset$.

```
public static void Example(object[] a) {
  Contract.Requires(a != null);
  for (var i = 0; i <= a.Length; i++) {
    a[i] = ...f(a[i])... ;  // (*)
    if (NonDet()) return;
  }
}
```

Fig. 1. The weakest precondition for this code is *false*, which rules out *good* executions. Our technique only excludes *bad* runs, inferring the necessary precondition $0 < \texttt{a.Length}$.

3 Sufficient Preconditions

The weakest (liberal) preconditions provide *sufficient* preconditions which guarantee the (partial) correctness, *i.e.*, the absence of errors in the program [2, 5, 9, 22, 29, 31]:

$$\forall s \in \Sigma : \mathsf{wlp}(\mathtt{P}, \mathsf{true})(s) \triangleq (\mathcal{E}(s) = \emptyset).$$

The main drawbacks preventing the use of the weakest (liberal) preconditions calculus for automatic precondition inference are: (i) in the presence of loops, there is no algorithm that computes weakest (liberal) precondition $\mathsf{wlp}(\mathtt{P}, \mathsf{true})$, (ii) due to loop over-approximation, the inferred preconditions are sufficient but no longer the weakest, and (iii) the resulting sufficient (liberal) preconditions may be too strong and rule out good runs.

More formally, an *under-approximation* $\overline{P}$ of $\mathsf{wlp}(\mathtt{P}, \mathsf{true})$ on states $s \in \Sigma$ must be computed such that

$$\begin{aligned}
& \forall s \in \Sigma : \overline{P}(s) \Rightarrow \mathsf{wlp}(\mathtt{P}, \mathsf{true})(s) && \wr \text{under-approximation} \int \\
\Leftrightarrow\ & \forall s \in \Sigma : \overline{P}(s) \Rightarrow (\mathcal{E}(s) = \emptyset) && \wr \text{def. } \mathsf{wlp}(\mathtt{P}, \mathsf{true}) \int \\
\Leftrightarrow\ & \forall s \in \Sigma : \overline{P}(s) \Rightarrow (\mathcal{T}(s) \neq \emptyset \vee \mathcal{I}(s) \neq \emptyset) \\
& \qquad \wr \text{since } \mathcal{S}(s) \neq \emptyset \wedge \mathcal{S} = \mathcal{E} \cup \mathcal{T} \cup \mathcal{I} \text{ so } \mathcal{E}(s) = \emptyset \text{ implies } (\mathcal{T}(s) \cup \mathcal{I}(s)) \neq \emptyset. \int \\
\Leftrightarrow\ & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [\overline{P}(s) \Rightarrow (\mathcal{T}(s) \neq \emptyset)] && (1)
\end{aligned}$$

The precondition $\overline{P}$ on states $s \in \Sigma$ is *sufficient* for the absence of definite runtime errors (under the termination hypothesis) but $\neg\overline{P}(s) \wedge (\mathcal{T}(s) \neq \emptyset)$ is possible so if execution stops in states s such that $\neg\overline{P}(s)$ (the sufficient precondition fails) then $\mathcal{T}(s) \neq \emptyset$ implies that other valid executions may be ruled out.

We argue that in general the use of a sufficient precondition is unfair in an automatic static analysis assume/guarantee reasoning setting, as: (i) it rules out correct executions; and (ii) it imposes too strong a proof obligation at call-sites.

Example 1 (Ruling out good runs). Consider the code in Fig. 1, a very simplified version of some pattern we found in `System.dll`. The function `NonDet` is a nondeterministic Boolean function. At runtime, an out-of-bounds array access at line $(*)$ may or may not appear. If the array is empty then the execution will fail

```
int Sum(int[] xs)
{
  Contract.Requires(xs != null);
  int sum = 0;
  for(var i = 0; i < xs.Length; i++)
    sum += xs[i];
  Assert(sum >= 0);
  return sum;
}
```

Fig. 2. In presence of overflows, the weakest precondition of `Sum` is essentially the method itself. The weakest precondition can be overapproximated (as customary in deductive verification) with a *sufficient* precondition. Automatically inferred sufficient preconditions may require the caller to establish a too strict condition.

while trying to access the first element. Otherwise, the failure will not appear if some of the `NonDet` calls returns `true`.

The weakest (liberal) precondition is *false*, meaning that all the runs, even the good ones, are rejected. To us, this is too strict. We propose a definition where no *good* run is removed, but only *bad* ones. For instance according to our definition, it is correct for our automatic tool to infer the precondition $\texttt{a.Length} > 0$ as an empty array will definitely lead to a failure at runtime. □

Example 2 (Requiring too much from the client). Let us consider the method in Fig. 2, where `int` is a 32-bit integer, and overflows are not an error, but a desired side-effect. `Sum` returns 19 for the input array $\{-2147483639, -2147483628, -10\}$. The weakest precondition of the method `Sum` is essentially the method itself:

$$\left(\overline{\sum}_{0 \le j < \texttt{xs.Length}} \texttt{xs[i]}\right) \ge 0. \tag{2}$$

It is a second order formula as $\overline{\sum}$, the sum on two's complement is defined inductively. The automatic inference of (2) is tough, out of reach of the current state-of-the-art tools and inference techniques. One can imagine tools inferring weaker loop invariants, originating in stronger sufficient preconditions. Two possible sufficient preconditions for the method are

$$\forall j \in [0, \texttt{xs.Length}).\ 0 \le \texttt{xs}[j] < \texttt{MaxInt/xs.Length} \tag{3}$$

$$\texttt{xs.Length} = 3 \wedge \texttt{xs}[0] + \texttt{xs}[1] = 0 \wedge \texttt{xs}[2] \ge 0. \tag{4}$$

as they satisfy the corresponding Hoare triples $\{(3)\}\ \texttt{C}\ \{Q\}$, $\{(4)\}\ \texttt{C}\ \{Q\}$ and $\{(3) \vee (4)\}\ \texttt{C}\ \{Q\}$. However, it is unfair to use one of them for an automatic modular assume/guarantee reasoning. For example, the input array above satisfies neither (3) nor (4). So, a tool inferring a sufficient but not necessary precondition will report a precondition violation for a caller with such an actual parameter. □

A sufficient precondition may impose too large a burden on callers, thereby making the precondition appear wrong to the user. In an early attempt at precondition inference, we were inferring sufficient but not necessary preconditions.

		$\mathcal{T}(s)$	
	$\overline{P}(s)$	$\emptyset$	$\neq \emptyset$
$\mathcal{E}(s)$	$\emptyset$	true	true
	$\neq \emptyset$	false	false

$\overline{P}(s) = \mathsf{wlp}(\mathtt{P}, \mathsf{true})(s) \triangleq (\mathcal{E}(s) = \emptyset)$

		$\mathcal{T}(s)$	
	$\underline{P}(s)$	$\emptyset$	$\neq \emptyset$
$\mathcal{E}(s)$	$\emptyset$	true	true
	$\neq \emptyset$	false	true

$\underline{P}(s) = (\mathcal{T}(s) \neq \emptyset \vee \mathcal{E}(s) = \emptyset)$

Fig. 3. (a) Weakest *sufficient* precondition (b) Strongest *necessary* precondition

Our users (professional programmers with no background in formal methods) filed several bug reports, marking such preconditions as *"wrong"* suggestions from `cccheck`.

4 Necessary Preconditions

We advocate the use of necessary preconditions, *i.e.*, preconditions which, once violated, definitely lead to an error later in the program execution. Such a *necessary* precondition $\underline{P}$ on states $s \in \Sigma$ is

$$\begin{aligned}
& \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [(\mathcal{T}(s) \neq \emptyset) \Rightarrow \underline{P}(s)] && \text{⟅inverse of (1)⟆} \\
\Leftrightarrow\ & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [\neg \underline{P}(s) \Rightarrow (\mathcal{T}(s) = \emptyset)] && \text{⟅contraposition⟆} \\
\Leftrightarrow\ & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [\neg \underline{P}(s) \Rightarrow (\mathcal{T}(s) = \emptyset \wedge \mathcal{E}(s) \neq \emptyset)] \\
& \qquad \text{⟅since } \mathcal{S}(s) \neq \emptyset,\ \mathcal{S} = \mathcal{E} \cup \mathcal{T} \cup \mathcal{I},\ \mathcal{I}(s) = \emptyset, \text{ and } \mathcal{T}(s) = \emptyset \text{ imply } \mathcal{E}(s) \neq \emptyset. \text{⟆}
\end{aligned}$$

If the necessary precondition $\underline{P}(s)$ does not hold then an execution from s either diverges or else it definitely terminates in an error (since $\mathcal{T}(s) = \emptyset$ so there is no possible finite correct execution). Setting apart infinite traces (*i.e.* $\mathcal{I}(s) = \emptyset$), Fig. 3 shows that the difference is only when $\mathcal{E}(s) \neq \emptyset \wedge \mathcal{T}(s) \neq \emptyset$. Whereas the sufficient precondition rules out all correct executions (since $\overline{P}(s) = \mathsf{false}$) the necessary precondition allows all of them (since $\underline{P}(s) = \mathsf{true}$), but maybe including erroneous ones.

5 Intra-procedural Precondition Inference

We briefly recall and illustrate three of the four algorithms introduced in [11] to infer under-approximations of necessary preconditions, that we have implemented in `Clousot/cccheck`, the static analyzer of CodeContracts. These fully automatic static analyses effectively compute preconditions with concretization $P^\flat$ such that

$$\forall s \in \Sigma : P^\flat(s) \Rightarrow (\mathcal{T}(s) = \emptyset \wedge \mathcal{E}(s) \neq \emptyset)\ .$$

These preconditions $P^\flat$ are therefore sufficient to guarantee the presence of definite errors (*i.e.* runtime error, programmer assertion or post condition failures) or non-termination. So $P^\flat$ is an under-approximation of the error semantics

```
void Partition(int[] array, int left, int right, int pivotIndex) {
 1:  var pivotValue = array[pivotIndex];
 2:  swap(ref array[pivotIndex], ref array[right]);
 3:  var storeIndex = left;
 4:  for (var i = left; i < right; i++)
 5:  {
 6:    if (array[i] < pivotValue) {
 7:      swap(ref array[i], ref array[storeIndex]);
 8:      storeIndex++; }
 9:  }
10:  swap(ref array[storeIndex], ref array[right]);
}
```

Fig. 4. A partitioning routine as found in QuickSort implementations. Our technique infers necessary preconditions (5) and (6) which turn out to be also sufficient.

$\mathcal{T}(s) = \emptyset \wedge \mathcal{E}(s) \neq \emptyset$ while its negation $\neg P^\flat$ is weaker than the strongest necessary precondition

$$\forall s \in \Sigma : (\mathcal{T}(s) \neq \emptyset \vee \mathcal{E}(s) = \emptyset) \Rightarrow \neg P^\flat(s) \ .$$

The inferred preconditions are therefore *necessary* in that they do not guarantee (in general) the correctness of the method, but if not established they certainly imply its failure. As shown by the experiments, these necessary preconditions may also be sufficient to prove the correctness of the assertion they originated from.

5.1 All-Paths Precondition Analysis (APPA)

The all-paths precondition analysis (APPA) of [11, Sect. 7] symbolically hoists assertions $\langle \mathtt{c}, \mathtt{b} \rangle \in \mathbb{A}$ all the way back to the code/method entry. Three conditions should hold: (i) the value of `b` is the same at `c` and at `entry`; (ii) the value of `b` is checked on all paths from the `entry`; and (iii) the variables in `b` have the correct visibility. In general, the generated precondition will be an atomic formula, containing a disjunction if and only if `b` contains one. From the three conditions above, it follows that if `b` holds at entry then it will also hold in all the method paths, therefore the generated precondition is also sufficient (for `b`).

Example 3 (Atomic preconditions (APPA)). Consider the in-place version of the partitioning phase of Quicksort (Fig. 4). In the example, `cccheck` verifies 20 assertions (null-pointer accesses, lower and upper array bounds) and issues 7 warnings. The warnings are the array dereference and the two array bounds checks at line `1`, two array bounds checks at line `2`, the lower bound check at line `6` and the upper bound check `array`[`storeIndex`] at line `10` (the analysis infers `storeIndex` $\geq$ `0` at line `10`). A necessary precondition has to be generated for those warnings. The `array` dereference at line `1` definitely causes an error if the actual value of `array` is null. Same for the array loads at lines `1` and `2`: if `pivotIndex` and `right` are not in the bounds of `array` then the program will fail for sure. These assertions can be pushed up to the entry point to generate the following *necessary* preconditions:

```
public void Combination(string x, int z) {
  Contract.Requires(z >= 0);
  while (z > 0)  { z--; }
  // here Clousot infers  z == 0
  if (z == 0)
    Assert(x != null);
}
```

Fig. 5. A simplified example from `mscorlib` where the information inferred by a forward static analysis is used to generate better preconditions

```
Contract.Requires(array != null && 0 <= pivotIndex);
Contract.Requires(pivotIndex < array.Length);
Contract.Requires(0 <= right && right < array.Length);        (5)
```

□

5.2 Conditional-Path Precondition Analysis (CPPA)

The conditional-path precondition analysis (CPPA) of [11, Sect. 9], hoists more assertions $\langle \mathtt{c}, \mathtt{b} \rangle \in \mathbb{A}$ to the procedure entry point than APPA by taking into account program paths and tests, and using dual widening to cope with infinite path lengths. The basic abstract predicates $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ mean that when the path condition $\mathtt{b}_p$ holds, execution will definitely be followed by an `assert(b)` and checking $\mathtt{b}_a$ at the beginning of the path is the same as checking this `b` later in the path when reaching the assertion. The partial order is $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a \Mapsto \mathtt{b}'_p \rightsquigarrow \mathtt{b}'_a \triangleq \mathtt{b}'_p \Mapsto \mathtt{b}_p \wedge \mathtt{b}_a \Mapsto \mathtt{b}'_a$ where the abstract implication $\mathtt{b} \Mapsto \mathtt{b}'$ underapproximates the concrete implication $\Rightarrow$: $\mathtt{b} \Mapsto \mathtt{b}'$ implies that $\forall s \in \Sigma : [\![\mathtt{b}]\!]s \Rightarrow [\![\mathtt{b}']\!]s$. In general, the analysis will generate atomic preconditions containing disjunctions. If no loops are encountered in the path between `entry` and `c`, then the generated necessary precondition is also sufficient (for `b`).

Example 4. In the `Combination` procedure of Fig. 5, `Clousot` infers that $\mathtt{z} = 0$ after the loop, so that the precondition $\mathtt{x} != \mathtt{null}$ is generated. Note that the invariant inferred from `Clousot` is crucial to infer the precondition (no precondition would be inferred otherwise). □

Example 5 (Atomic preconditions (CPPA)). Continuing Ex. 3, we are left with two candidate assertions. In one case the assertion is not checked on every path (line 6, unreached if `left`$\geq$ `right`). In the other case `storeIndex` may have been modified (line 10). So the APPA analysis above does not apply. With CPPA, the candidate assertions are propagated backwards, taking into account tests and computing a fixpoint. We can then infer the disjunctive necessary preconditions:

```
Contract.Requires(left < right  || left < array.Length);
Contract.Requires(left >= right || 0 <= left);                (6)
```

Informally, the two preconditions state that whenever `left` $<$ `right` then `left` should be non-negative otherwise `left` $<$ `array.Length`.

```
void ReadAndConsume(Message[] msg) {
1: Contract.Requires(msg != null);
2: for (var i = 0; i < msg.Length; i++) {
3:   Assert(msg[i] != null);
    // Do something with msg[i], then consume it
4:   msg[i] = null;
   }
  // Here msg[*] == null
}
```

Fig. 6. To prevent an error, `msg` should not contain any `null` values. Inferring this precondition (7) requires non-trivial reasoning about which elements of the array have been tested and modified.

In this case, the inferred necessary preconditions are also sufficient, as it can be easily checked by instrumenting `Partition` with the inferred preconditions and running `cccheck` again. Please note that the preconditions above are weaker than the usual ones found in the specification of the partition algorithm which require $0 \leq$ `left` and `left` $<$ `right` to ensure functional correctness (*e.g.*, in Hoare's original paper on QuickSort [24]). □

5.3 Quantified Precondition Analysis (QPA)

The APPA and CPPA analyses cannot deal directly with unbounded data structures such as collections and arrays. [11, Sect. 10] uses a forward static analysis based on [12] to synthesize quantified preconditions. This quantified precondition analysis (QPA) can deduce that a subset of the collection elements are: (i) checked in every execution path; and (ii) when checked, they have the same value they had at entry, so as to synthesize a *universally* quantified precondition.

Example 6 (Universally quantified preconditions). The method precondition for the example in Fig. 6 is too weak to prevent a runtime error: if `msg` is not empty and one of its elements is `null` then the program will definitely fail at runtime. The precondition should be quantified over the array elements, so the inference of atomic preconditions above does not help. The inference is non-trivial as the content of the input array is modified inside the loop: The analysis should make sure that the checked array elements are the same as in the pre-state. We infer the necessary universally quantified precondition

```
Contract.Requires(ForAll(message, msg => msg != null));
```
(7)

Please note that: (i) the precondition encompasses the case of an empty array; and (ii) it is also sufficient. □

Inferred necessary preconditions may not be sufficient, in particular when the condition must be weakened during backward propagation due to control flow or loops.

Example 7 (Necessary but not sufficient preconditions). Consider the code in Fig. 7, a simplified version of a common pattern used in C/C++ programs

```
int FirstOccurrence(int[] a) {
  Contract.Requires(a != null);
  var i = 0;
  while (a[i] != 3) { i++; }
  return i;
}
```

Fig. 7. An example where all the inferred atomic preconditions are necessary but not sufficient. A sufficient existentially quantified precondition (8) can be inferred by a forward array content and modification analysis.

and .NET framework libraries. Pushing the array index condition backwards produces the precondition

$$\texttt{Contract.Requires}(\texttt{a.Length} > 0),$$

which is not sufficient. If we semantically unroll the loop k times, we can generate increasingly stronger necessary preconditions of the form

$$\texttt{a.Length} > 0 \wedge \texttt{a}[0] = 3 \vee \texttt{a.Length} > 1 \wedge \texttt{a}[1] = 3 \vee \ldots,$$

yet none of them are sufficient. In general, the precondition inference requires a fixpoint computation over an infinite domain. The convergence of the computation should be enforced using a widening operator. In the weakest precondition calculus, using a widening can very easily bring to the inference of sufficient preconditions. In necessary precondition inference, the *dual* widening can simply stop the iterations after k iterations — A widening over-approximates its arguments, while a dual widening under-approximates them. The desired precondition is existentially quantified:

$$\exists j : j \in [0, \texttt{array.Length}) \wedge \texttt{array}[j] = 3 \tag{8}$$

Such a precondition can be inferred by combining forward array content and array modification analyses. □

6 Scaling Up Thanks to Simplification

The disjunctive precondition $P^\flat$ represented as a set $\mathcal{P}_\mathbb{A}$ of terms in $\mathbb{A}$ may contain redundant preconditions which should be removed for two main pragmatic reasons. First, the more preconditions, the more proof obligations need to be discharged in other methods. Second, more preconditions mean more suggestions to the end-user, who may get irritated if they are redundant (as we experienced with `cccheck`).

We would like to compute a minimal yet equivalent set $\mathcal{P}^m_\mathbb{A}$. The set $\mathcal{P}^m_\mathbb{A}$ should be: (i) a set of generators ($\forall \mathtt{p} \in \mathcal{P}_\mathbb{A} : \exists \{\mathtt{p}_0 \ldots \mathtt{p}_n\} \subseteq \mathcal{P}^m_\mathbb{A} : \mathtt{p}_0 \wedge \cdots \wedge \mathtt{p}_n \Rightarrow \mathtt{p}$); and (ii) minimal ($\forall \mathtt{p} \in \mathcal{P}_\mathbb{A} : \bigwedge \{\mathtt{q} \mid \mathtt{q} \in \mathcal{P}^m_\mathbb{A} \setminus \{\mathtt{p}\}\} \not\Rightarrow \mathtt{p}$). Unfortunately, computing a minimal set of generators can be very expensive [15] or even impossible when the inferred invariants are quantified. To see why, let $\mathtt{p}_1$ and $\mathtt{p}_2$ be two Boolean expressions containing quantified facts over arrays, then $\mathtt{p}_1 \Rightarrow \mathtt{p}_2$ is not decidable [3]. There exist subsets for which the problem is decidable, *e.g.*, equalities or difference constraints. In general the minimal set of generators is not unique.

$\mathsf{simpl} \triangleq \boldsymbol{\lambda}\, \mathcal{P} \cdot$

$\texttt{true} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{true}\}$

$\texttt{true}||\texttt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{true}||\texttt{b}\}$ (|| is the non-commutative short-cutting disjunction)

$\texttt{false} \in \mathcal{P} \quad \rightarrow \quad \{\texttt{false}\}$

$\texttt{false}||\texttt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{false}||\texttt{b}\} \cup \{\texttt{b}\}$

$\texttt{b}||t \in \mathcal{P} \wedge t \in \{\texttt{true}, \texttt{false}\} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{b}||t\} \cup \{t||\texttt{b}\}$

$\texttt{b}_1||\texttt{b},\ \texttt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{b}_1||\texttt{b}\}$

$\texttt{b}_1||\texttt{b},\ !\texttt{b}_1||\texttt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{b}_1||\texttt{b},\ !\texttt{b}_1||\texttt{b}\} \cup \{\texttt{b}\}$

$\texttt{b}_1||\texttt{b},\ \texttt{b}_2||\texttt{b} \in \mathcal{P} \wedge s(\texttt{b}_1) \subseteq s(\texttt{b}_2) \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{b}_2||\texttt{b}\}$

$\texttt{f} \triangleq \forall i \in [\texttt{l}, \texttt{l}+\texttt{1}) : \texttt{b}(i) \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{f}\} \cup \{\texttt{b}(\texttt{l})\}$

$\texttt{f}_1 \triangleq \forall i \in [\texttt{l}, \texttt{v}) : \texttt{b}(i), \texttt{f}_2 \triangleq \forall i \in [\texttt{v}+\texttt{1}, \texttt{u}) : \texttt{b}(i) \in \mathcal{P}$
$\quad \rightarrow \quad \mathcal{P} \setminus \{\texttt{f}_1, \texttt{f}_2\} \cup \{\forall i \in [\texttt{l}, \texttt{u}) : \texttt{b}(i)\}$

$\texttt{x}_1 - \texttt{x}_2 \leq v_1, \texttt{x}_2 - \texttt{x}_3 \leq v_2, \texttt{x}_1 - \texttt{x}_3 \leq v_3 \in \mathcal{P}$
$\quad \rightarrow \quad$ *if* $v_1 + v_2 \leq v_3$ *then* $\mathcal{P} \setminus \{\texttt{x}_1 - \texttt{x}_3 \leq v_3\}$ *else* $\mathcal{P}$

Fig. 8. The simplification for the candidate preconditions. The function $s(\texttt{b})$ returns a set whose constraints are the conjuncts in $\texttt{b}$.

In practice, we are not interested in getting the best $\mathcal{P}^m_{\mathbb{A}}$, but only a good approximation. The approximation should be such that: (i) obviously-redundant preconditions are removed; and (ii) it is fast, in that only an infinitesimal fraction of the analysis time allocated for the procedure is spent on the simplification. In our implementation we use a simple heuristics to simplify the candidate preconditions and get a set $\overline{\mathcal{P}}^m_{\mathbb{A}} \supseteq \mathcal{P}^m_{\mathbb{A}}$ such that $\#\overline{\mathcal{P}}^m_{\mathbb{A}} \leq \#\mathcal{P}_{\mathbb{A}}$, for some minimal set of generators $\mathcal{P}^m_{\mathbb{A}}$.

The simplification equations are given in Fig. 8. The rationale is that we want to simplify as many disjunctive preconditions as possible, trivial quantified facts, and difference constraints. The precondition `true` or any disjunct containing it can be eliminated from the set. If `false` appears as an atom, then there is no way to satisfy the precondition. `false` is the identity for disjunction, so it can be cancelled. In general the language short-cutting disjunction || is not commutative, but in our simplification procedure it can be moved to the front position (to enable the previous two rules). If an expression `b` is already in $\mathcal{P}$ with no antecedent, we can safely remove all the preconditions where `b` appears as a consequence. When `b` is implied by some condition and by its negation, then we can simply remove the conditions (we found this being a very common case in practice, when the precondition does not depend on the paths through a conditional statement). For remaining pairs with the same conclusion, we only retain the (disjunctive) precondition with fewer hypotheses. As for quantified facts, we remove those that boil down to a singleton and we merge together consecutive intervals. Finally, we remove from the difference constraints those that are redundant. The simplification should be iterated to a fixpoint: $\mathsf{Simplify}(\mathcal{P}_{\mathbb{A}}) \triangleq \mathsf{simpl}^*(\mathcal{P}_{\mathbb{A}})$.

```
Next ← ProceduresOf(P)
while Next ≠ ∅ do
  m ← PickOneProcedure(Next)
  𝔸 ← SpecificationFor(m, Clousot)
  𝒫_𝔸 ← InferPrecondition(𝔸, Clousot)
  P̄^m_𝔸 ← Simplify(PreconditionsOf(m) ∪ 𝒫_𝔸)
  if P̄^m_𝔸 ≠ PreconditionsOf(m) then
    PreconditionsOf(m) ← P̄^m_𝔸
    Next ← (Next \ {m}) ∪ callersOf(m)
  else
    Next ← Next \ {m}
  end if
end while
```

Fig. 9. The inter-method preconditions inference algorithm

7 Inter-procedural Precondition Inference

The inter-procedural precondition inference algorithm is shown in Fig. 9. The input program P can either be a complete program or a library. We assume that each procedure m has an initial set of preconditions $\mathsf{PreconditionsOf}_0(\mathtt{m})$, which can be empty. The set Next contains the procedures to be analyzed (continuation set). It is initialized with all the procedures in the program P.

In the loop body, we first pick a procedure m from the continuation set. We leave the implementation of the policy PickOneProcedure as a parameter of the inference algorithm. For instance PickOneProcedure may be the bottom-up traversal of P's call graph.

The function SpecificationFor returns the residual specification $\mathbb{A}$ of m by running Clousot to discharge as many proof obligations as possible. If $\mathbb{A} = \emptyset$, then we say that the preconditions PreconditionsOf(m) are *sufficient* to ensure that each proof obligation in m is either always correct or always wrong. The analyzer Clousot is left as a parameter.

We infer the set $\mathcal{P}_{\mathbb{A}}$ of necessary preconditions from the assertions in $\mathbb{A}$. The intra-method inference algorithm InferPrecondition is one of the previously described analyses such as APPA, CPPA, or QPA.

Next, the function Simplify removes redundant preconditions so as to obtain an approximate minimal set $\overline{\mathcal{P}}^m_{\mathbb{A}}$ of conditions as described in Sec. 6.

Finally, if we discovered new preconditions for m, we add them to the set of its preconditions, and we update the continuation set by adding all the callers of m, which we must re-analyze due to the stronger proof-obligations at these call-sites. In case we discovered no new precondition, we simply remove m from the continuation set.

The inter-method inference algorithm in Fig. 9 is a least fixpoint computation on the abstract domain $\langle \mathsf{ProceduresOf}(\mathtt{P}) \to \wp(\mathbb{B}), \dot{\Leftarrow} \rangle$, where $\mathbb{B}$ is the set of side-effect free Boolean expressions and $\dot{\Leftarrow}$ is a sound abstraction of the pointwise

```
void f(string x, string y) {
  Assert(x != null);
  g(x, y);
}
```

```
void g(string u, string v) {
  Assert(v != null);
  f(u, v);
}
```

Fig. 10. An example of inference of preconditions for mutually recursive methods requiring a fixpoint computation

inverse logical implication. In the presence of (mutual) recursion the algorithm may not terminate: for instance it may infer the increasing chain of preconditions $\{0 < \mathtt{a}\} \dot{\Leftarrow} \{1 < \mathtt{a}\} \dot{\Leftarrow} \ldots$ To enforce convergence, a dual widening operator should be used: the simplification is incomplete so it does not solve the convergence problem in the abstract even in case of convergence in the concrete. Easy dual widening operators are either bounding the maximum number of times a method is analyzed or the maximum cardinality of $\mathsf{PreconditionsOf}(\mathtt{m})$ or both. Ignoring preconditions is safe: intuitively it means that fewer checks are pushed up in the call stack but warnings are still reported in the callee.

In practice, we can stop inferring new necessary preconditions at any point. The remaining methods in `Next` then simply need to be checked again (without inferring new preconditions) to obtain the final set of warnings.

Example 8. Let us consider the two mutual procedures in Fig. 10. Ignore non-termination: we choose a minimalistic example to illustrate the inter-method fixpoint computation. Let us suppose `f` is the first method to be picked up. The intra-method precondition inference algorithm obtains $\mathsf{PreconditionsOf}(\mathtt{f}) = \{\mathtt{x} = \mathtt{null}\}$. The preconditions for `g` are then $\{\mathtt{u} = \mathtt{null}, \mathtt{v} = \mathtt{null}\}$. The procedure `f` is a caller of `g`, so it is added to the continuation set. The re-analysis is enough to reach the fixpoint: $\mathsf{PreconditionsOf}(\mathtt{f}) = \{\mathtt{x} = \mathtt{null}, \mathtt{y} = \mathtt{null}\}$. □

7.1 Provenance

Each precondition `p` in $\mathcal{P}_{\mathbb{A}}$ originates from at least one failing proof obligation $\langle \mathtt{c}, \mathtt{b} \rangle \in \mathbb{A}$. We can construct a provenance relation $\mathtt{p} \gg \mathtt{b}$, with the intuitive meaning that if `p` does not hold at the method entry, then `b` will fail later. We use the provenance chain $\mathtt{p}_{n-1} \gg \cdots \gg \mathtt{p}_0 \gg \mathtt{b}$ to report an inter-method error trace to the user. Furthermore, we can suppress the warning for `b` if we detect that `p` is also sufficient to prove `b` safe, *i.e.*, `p` holds at the entry point if and only if `b` holds at program point `c`. This is the case when at least *one* of these conditions holds: (i) the method `m` does not contain loops; (ii) `p` is inferred using APPA (Sec. 5.1); (iii) `p` is inferred using CPPA (Sec. 5.2) *and* no loop is encountered in the path between `entry` and `c`. Essentially, if we detect that the generated necessary precondition `p` is also sufficient to discharge `b`, we can push the *full* burden of the proof to the callers of `m`. Otherwise, we report the warning to the user and we propagate `p` to the callers, as failure to establish it will definitely cause an error in `b`: by propagating `p` we make explicit to the callers thay they *must* establish `p` before calling `m`.

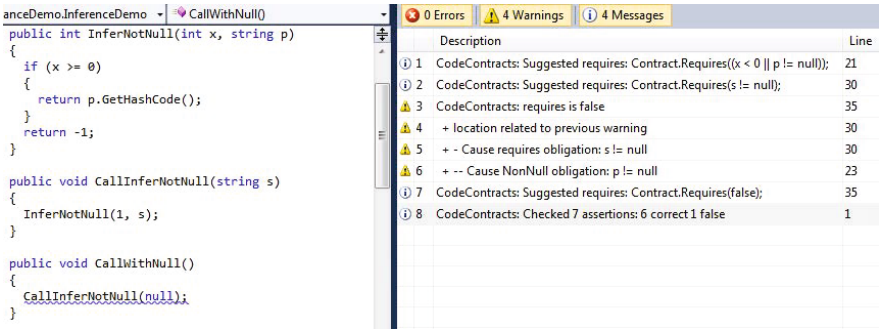


Fig. 11. A screenshot of the error reporting with the precondition inference

8 Experience

We implemented the analyses described above in `cccheck`, an industrial-strength static analyzer and contract checker for .NET bytecode based on abstract interpretation. It uses abstract interpretation to infer facts about the program and to discharge the proof obligations. `cccheck` performs a modular assume/guarantee analysis: for each method it assumes the precondition and it asserts the postcondition. At method calls, it asserts the precondition, and it assumes the postcondition. It performs a simple postcondition inference to propagate whether a method returns a non-null value and the expression returned by getters. The necessary precondition inference is enabled by default in every run of the analyzer, and used by our customers since June 2011 on a daily basis. The kind of intra-method precondition inference algorithm can be selected by a command line switch. The all-path precondition analysis (APPA) is the default. We implemented the analyses faithfully respecting the formalization in the previous sections. The only differences are in the quantified precondition analysis: (i) we restrict it to arrays of objects (instead of collections of generic types); (ii) the only assertions we check for are not-null; and (iii) the quantified preconditions are suggested to the user but not propagated to the callers (yet).

The main motivation for this work was to help our users getting started with `cccheck`, by suggesting preconditions, so that users can simply add them to their code (or automatically with support from the IDE).

In an early stage of this work, we had a simple analysis to infer *sufficient* preconditions (Sect. 3). Essentially, if we could prove that the value of an unproven proof obligation was unchanged from the entry point, then we suggested the corresponding expression as a precondition. This is similar to the ESC/-Java `/suggest` switch [18]. The problem with this naïve approach was that it did not take into account tests and different execution paths and so can also eliminate good runs. The result was confusing for our customers, who filed several bug reports about "wrong" suggestions (essentially preconditions that were too strong according to the user). For instance, in the code of Fig. 11 our old

analysis would have produced the too strong precondition `p!=null` for the method `InferNotNull`.

Necessary precondition inference reduces the warning noise of the analyzer and raises the automation bar by providing inter-method propagation of preconditions. If a necessary precondition can be inferred from a proof obligation and `cccheck` determines that it is also sufficient then the warning can be omitted, i.e., the full burden of establishing it is pushed to the callers. The check for sufficiency can be: (i) by construction, *e.g.*, when APPA is used; (ii) inferred from the analysis, *e.g.*, by detecting that the dual widening in CPPA, if any, has introduced no loss of precision; (iii) by re-analysis, the generated precondition is injected in the method which is then reanalyzed. In `cccheck` we use (i) and (ii). We instrumented `cccheck` to perform (iii) to collect the data for Sec. 8.3.

Example 9. Let us consider the screenshot in Fig. 11 showing the result of a run of `cccheck` inside Visual Studio. On the right is the list of suggestions and warnings produced by `cccheck` for the code on the left. The analyzer performs a bottom-up analysis, based on a precomputed approximation of the call graph. First it analyzes `InferNotNull`. It determines that when `x` is non-negative and `p` is `null` then a runtime error may occur. It infers a necessary precondition (message #1) which is also sufficient—no approximation comes from loops. Therefore the generated precondition fully captures the safety requirement and no warning is issued to the user for `p`'s dereference since the proof burden is pushed to the callers—in our example `CallInferNotNull`. This method does not fully establish precondition #1, and our inference computes another necessary precondition #2. Please note that this precondition is simpler than #1 since our simplification procedure removed the trivial disjunct $1 < 0$. Precondition #2 happens to also be sufficient in this case as there are no loops. `cccheck` detects this fact, and no warning is issued in method `CallInferNotNull`. Instead, precondition #2 is propagated to the call-site within `CallWithNull`. There, `cccheck` determines that the call does not satisfy the inferred necessary precondition of `CallInferNotNull`. It reports the error message (#3) as well as the inter-method provenance trace (messages #4 . . . #6). The generated precondition #7 encodes the fact that all invocations of `CallWithNull` will definitely cause an error. □

8.1 Benchmarks

We report our experience on two different sets of benchmarks. The first one contains industrial libraries without existing contracts. We have chosen the latest versions of `mscorlib.dll`, `System.dll` and `System.Core.dll` as they are the main libraries of the .NET framework, and `System.Data.dll` because, in our experience, it causes trouble for static analyzers. The first three libraries contain the most common shared functionalities of the .NET framework (system type definitions, collections, reflection, cryptographic and communication primitives, etc.). The last one contains common code to access data from diverse sources. `cccheck` analyzes bytecode, so we can use it directly on the shipped binaries. The libraries are in every copy of Windows under the directory `Windows/Microsoft.Net/Framework/`. For our experiments we run `cccheck`

Library	All-paths APPA $\mathcal{P}_{\mathbb{A}}$	$\overline{\mathcal{P}}^m_{\mathbb{A}}$	% rem.	Conditional-path CPPA $\mathcal{P}_{\mathbb{A}}$	$\overline{\mathcal{P}}^m_{\mathbb{A}}$	% rem.	QPA $\forall$
`mscorlib`	5133	3437	33.04%	8756	6564	25.03%	36
`System`	4409	3446	21.84%	6709	5533	17.53%	9
`System.Core`	3202	2197	31.39%	4723	3744	20.73%	32
`System.Data`	5899	3563	39.60%	8435	5642	33.11%	11
Total	18643	12643	32.18%	28623	21483	24.94%	88
`Facebook`	146	119	18.49%	171	145	15.20%	1
`Facebook.Web`	53	53	0.00%	86	86	0.00%	0
`Facebook.Web.Mvc`	49	31	36.73%	25	10	60.00%	0
Total	248	203	18.15%	282	241	14.54%	1

Fig. 12. The number of inferred preconditions, for APPA, CPPA, and QPA, and the percentage of redundant preconditions removed by the simplification routine Simplify

with the default checks: non-null, array-out-of bounds obligations and contracts. Since the version of the libraries we analyzed contained no contracts, the only contracts are the inferred necessary preconditions propagated by `cccheck` itself.

The second benchmark is the open-source `C#` Facebook SDK which is available for download at `facebooksdk.codeplex.com`. It contains a set of libraries to help .NET programmers (Windows, Silverlight, Windows Phone 7, etc.) integrate their application with Facebook. We selected it because the code base is almost completely annotated with contracts. In our experiments, we opened the solution containing the SDK, built the source as-it-is and let `cccheck` run with the same settings as in the distribution: the collected proof obligations are non-null, array-out-of bounds, arithmetic errors, redundant assumptions detection, and the explicit contracts. We only added switches to force the analyzer to collect the data reported in the tables.

8.2 Inferred Necessary Preconditions

The table in Fig. 12 reports the number of necessary preconditions inferred for the benchmarks for all three analyses (APPA, CPPA, and QPA), when the fixpoint of the inter-method inference algorithm has been reached.

For the system libraries, the all-paths analysis (APPA) infers 18,643 necessary preconditions. The simplification step removes more than 32% of the candidates, resulting in 12,643 necessary preconditions that are suggested and propagated. For the Facebook SDK, APPA infers 248 candidates, filtering only 45.

The conditional-path analysis (CPPA) infers roughly 69% additional necessary preconditions than APPA for the system libraries but only 18% more for the Facebook SDK. The price to pay for the more refined analysis is time: in our experience CPPA can be up to 4 times slower than APPA. However, at worst the total inference time (including simplification) is less than 4 minutes for very complex libraries (`System.Data`). Otherwise the overall running time is on the order of tenths of seconds.

Library	# of methods total	at least one warn.	inferred nec. pre.	inferred suff. pre.	% inferred	% suff.	precision improve-ment	# m. with 0 warns
`mscorlib`	21226	6663	2765	1519	41.50%	22.80%	7.15%	16082
`System`	14799	5574	2684	1378	48.15%	24.72%	9.31%	10603
`System.Core`	5947	2669	1625	765	60.88%	28.66%	12.8%	4043
`System.Data`	11492	4696	2388	1152	50.85%	24.53%	10.02%	7948
Total	53464	19602	9462	4814	48.27%	24.56%	9.00%	38676
`Facebook`	455	186	111	93	59.68%	50.00%	20.43%	362
`Facebook.Web`	194	57	30	18	52.63%	31.58%	9.27%	155
`Facebook.Web.Mvc`	92	40	29	26	72.50%	65.00%	28.26%	78
Total	741	283	170	137	60.07%	48.41%	18.48%	595

Fig. 13. The experimental results for the all-paths precondition analysis (APPA)

Library	# of methods total	at least one warn.	inferred nec. pre.	inferred suff. pre.	% inferred	% suff.	precision improve-ment	# m. with 0 warns
`mscorlib`	21226	7107	4062	1811	57.15%	25.48%	8.53%	15930
`System`	14799	5759	3546	1576	61.57%	27.37%	10.64%	10616
`System.Core`	5947	2740	2104	810	76.79%	29.56%	13.62%	4017
`System.Data`	11492	4824	3280	1292	67.99%	26.78%	11.24%	7960
Total	53464	20430	12992	5489	63.59%	26.87%	10.26%	38523
`Facebook`	455	186	130	92	69.89%	49.46%	20.22%	361
`Facebook.Web`	194	110	80	61	72.73%	55.45%	31.44%	145
`Facebook.Web.Mvc`	92	23	8	5	34.78%	21.74%	5.43%	74
Total	741	319	218	158	68.34%	49.53%	21.32%	580

Fig. 14. The experimental results for the conditional path precondition analysis (CPPA)

We manually inspected the necessary preconditions inferred for the Facebook SDK to check whether the simplification algorithm left any redundancy. We found only one redundant precondition, inferred by CPPA for `Facebook.Web`.

As one may expect, we inferred fewer universally quantified necessary preconditions. We inspected the 36 quantified necessary preconditions inferred for `mscorlib.dll`. It turns out that the analysis inferred conditions that at first looked suspicious. After a deeper investigation, we found that the analysis was right. However, it is more difficult to judge whether the analysis missed necessary preconditions it should have inferred. So, we inspected the proof obligations for the Facebook SDK. In `Facebook` we found only two proof obligations requiring a quantified contract. However, the needed contracts are not preconditions but an object invariant and a postcondition. The other two libraries make little use of arrays, so there was nothing interesting to be inferred there. Overall, we found the quantified necessary precondition analysis precise and fast enough.

8.3 Quality of the Inferred Preconditions

We are left with the problem of judging the *quality* of the inferred necessary preconditions. Counting the number of inferred preconditions is not a good measure of success. Measuring the number of warnings without inference and with inference is a better approach, but can also be misleading for the following reason: if a method contains a warning and is called by n other methods, then if that single warning can be turned into a necessary precondition, it potentially results in n warnings at all call-sites. We decided to measure how the inference of necessary preconditions reduces the number of methods for which we report warnings. If we reduce the number of methods with warnings, we say that we improved the precision of our analyzer `cccheck`.

The tables in Fig. 13 and Fig. 14 report the effects of the all-path (APPA) and the conditional-path (CPPA) analyses on the precision of `cccheck`. For each library we report (i) the total number of methods analyzed; (ii) the number of methods on which `cccheck` originally reports at least one warning (without any inference); (iii) the number of methods for which `cccheck` infers at least one necessary precondition; and (iv) the number of methods for which the necessary preconditions were also sufficient to reduce the warnings in that method to zero. The next three columns indicate (i) the fraction of methods with inferred necessary preconditions; (ii) the fraction of these for which the inferred preconditions are also sufficient; and (iii) the precision improvement as an increase in the number of methods with zero warnings. The last column contains the final total number of methods with 0 warnings, *i.e.* the methods which either did not require the inference of any precondition, *plus* the methods for which the inferred necessary precondition is sufficient. To check whether the necessary preconditions are also sufficient, we check: (i) that we inferred some necessary precondition for the method; and (ii) that $\mathbb{A} = \emptyset$ after re-analysis.

The conditional-path precondition analysis (CPPA) infers far more preconditions than APPA, and in general those preconditions are also more complicated, because they can be disjunctive. As a consequence, it is not a surprise that the final number of methods with zero warnings is smaller in the case of CPPA: the additional warnings are generated by the propagated inferred preconditions that cannot be proven by `cccheck` at call-sites.

In the framework libraries we were able to infer a necessary precondition for 48% of methods with APPA and for 63% of methods with CPPA. Interestingly, the necessary preconditions turned out to be also sufficient in 25% of methods for either analysis. The precision is even better for the Facebook SDK where we inferred necessary preconditions for more than 60% of methods. Additionally, the necessary preconditions where sufficient to prove the method correct in almost 50% of cases! We manually inspected the methods with remaining warnings. These resulted from missing contracts, *e.g.* postconditions for interface calls and abstract methods and object invariants.

Overall, precondition inference improved the precision of `cccheck` by roughly 10% for the framework assemblies and roughly 20% for the Facebook SDK.

9 Related Work

Our objectives are hardly comparable with those of unsound tools like PREfix [4] or its lightweight version PREfast which filter out potential errors, bug-finding tools like SAGE [20] that extends systematic dynamic test generation with formal methods, property checking tools combining static analysis and testing like YOGI [30], or verification environments like VCC [14] because an unsound assertion or a series of failing tests cannot be used as formal specifications and automatic inference of program properties is much more difficult than static verification of given properties annotating programs. However, automatically inferred necessary preconditions would definitely be useful for all these tools.

The closest related work we are aware of is Success typings [26], where types of function arguments are used instead of first-order formulae to express preconditions. Similar to our work, success typings capture as types, the conditions under which a function will definitely cause a runtime type error. Their analysis is limited to runtime type errors as opposed to general assertions, and appears to be used only to detect definite bugs in an untyped language. In addition, our approach can be used to reduce the annotation burden in verification, and the expressiveness of our preconditions is more general.

Most of the work on precondition inference focuses on the inference of *sufficient* preconditions for the partial correctness of a module, *e.g.*, [2, 5, 9, 29, 31], even if it is never explicitly expressed in those terms. We have discussed in Sec. 3 the drawbacks of using sufficient precondition inference in an automatic analysis setting. We are not aware of papers on sufficient precondition inference experimentally validating the inferred preconditions: (i) at call-sites; and (ii) on as large a scale as we did.

The related problem of generating procedure summaries (to scale up whole program analyses) received much attentions. Summaries can either be obtained via a combination of heuristics and static analyses [17, 23], or via firm semantic and logical grounds, *e.g.*, [6, 9, 22]. However, the practical effectiveness of the so-inferred preconditions is still unclear: *e.g.*, [6] reports the inference of thousands of contracts but their quality—for instance to prove a property of interest like memory safety—is unknown.

Some of these approaches are also less modular than ours. E.g., the Houdini approach [17] starts out by adding a set of candidate preconditions (generated from a template) to all methods and then uses repeated sound modular reasoning to reject candidates that lead to unproven assertions. This approach produces preconditions that are non-necessary (no assertion would be triggered if violated). Worse, the set of preconditions produced for a method depends on the contexts that call the method. The fewer such contexts exist, the stronger the inferred precondition (in the worst case *false*). Our approach is more modular. The necessary preconditions inferred for a method do not depend at all on how the method is called.

Some authors focused on inferring preconditions of a fixed template [21, 34] or summaries on abstract domains of finite height [33, 36]. Those approaches to precondition inference inherit the problems of the techniques they are based on.

Templates require too much user attention (to provide the template), are brittle, and do not scale up. Finite height abstract domains are not powerful enough [8]. We do not make any of those hypotheses in this work.

SnuggleBug [7], the dual analysis of [32], as well as other authors [19, 25, 35] use under-approximating symbolic backwards or dynamic analyses to find bugs. Our work is different in that we start out with a program verification problem with a number of unproven assertions. We use necessary preconditions to reduce the annotation burden experienced by the programmer. Using our static analysis, we can tell when a method has no more errors, whereas bug finding cannot. At the same time, our approach can also expose definite bugs, as shown in the scenario of Fig. 11. A main difference of our work with all the above is the handling of loops via fixpoints rather than some finite, partial unrolling.

For instance, with a (minor) modification of the universally quantified forward analysis, the implementation infers the necessary precondition $\texttt{newCarsOnly} \Rightarrow \forall \texttt{x} \in \texttt{c} : \texttt{x.getYear}() = 2009$ for the running example of [7], requiring that all cars in a list are manufactured in 2009. Snugglebug unrolls loops once or twice and will find violations only if it can find a list where the first or second car has the wrong date, but not the third.

Necessary preconditions are needed to define the problem of extract method with contracts [13]. They can be extended for the inference of necessary object invariants [1] and to propose automatic and verified code repairs [27].

Finally, our inferred necessary preconditions are human readable and can be persisted into code as documentation and to help future analysis runs, whereas the intermediate state of bug finding tools is rarely in a form that would make it useful for human consumption.

10 Conclusions

We presented the design and implementation of a system for the inference of *necessary* preconditions. We illustrated the algorithm for the inter-procedural inference. The algorithm is parameterized by the static analyzer used to discharge the proof obligations, the intra-method inference analysis, and the candidate precondition simplification routine. We improved the intra-method analyses of [11] by refining them with the invariants inferred by the static analyzer. We have implemented the inference in `cccheck`, an industrial static contract checker, and evaluated the analysis on real industrial code. We showed that our simplification algorithm, even if not complete, performs very well in practice by removing up to 33% of redundant preconditions and finding only one case in which a redundant precondition was not removed. We were able to infer necessary preconditions for up to 60% of methods reporting some warning. We validated the quality of the inferred preconditions by checking whether they were also sufficient to remove *all* warnings in a method (not only the warning they originated from). This was the case for 25% of methods with warnings. Overall, the necessary precondition inference has a large positive impact on `cccheck`, by improving its precision (in terms of methods with no warnings) up to 21%.

Looking forward, we want to investigate necessary preconditions inference for program optimization (by factoring and pushing up inevitable checks) and extend the approach to the inference of necessary postconditions (by pushing the assertions back to method calls).

Acknowledgments. Leonardo De Moura for the discussions on simplification. Work supported in part by the National Science Foundation under Grant No. 0926166.

References

[1] Bouaziz, M., Fahndrich, M., Logozzo, F.: Inference of necessary field conditions with abstract interpretation. In: APLAS. LNCS (2012)
[2] Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: PLDI, pp. 46–55 (1993)
[3] Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
[4] Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. Softw., Pract. Exper. 30(7), 775–802 (2000)
[5] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Footprint Analysis: A Shape Analysis That Discovers Preconditions. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 402–418. Springer, Heidelberg (2007)
[6] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009)
[7] Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: PLDI, pp. 363–374 (2009)
[8] Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: PLILP, pp. 269–295 (1992)
[9] Cousot, P., Cousot, R.: Modular Static Program Analysis. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 159–178. Springer, Heidelberg (2002)
[10] Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: POPL, pp. 245–258 (2012)
[11] Cousot, P., Cousot, R., Logozzo, F.: Precondition Inference from Intermittent Assertions to Contracts on Collections. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 150–168. Springer, Heidelberg (2011)
[12] Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, pp. 105–118 (2011)
[13] Cousot, P., Cousot, R., Logozzo, F., Barnett, M.: An abstract interpretation framework for refactoring with application to extract methods with contracts. In: OOPSLA. ACM (2012)
[14] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: ICSE Companion, pp. 429–430 (2009)
[15] Dillig, I., Dillig, T., Aiken, A.: Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 236–252. Springer, Heidelberg (2010)
[16] Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)

[17] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for eSC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
[18] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: PLDI, pp. 234–245 (2002)
[19] Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)
[20] Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM 55(3), 40–44 (2012)
[21] Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-Based Invariant Inference over Predicate Abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2009)
[22] Gulwani, S., Tiwari, A.: Computing Procedure Summaries for Interprocedural Analysis. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)
[23] Hackett, B., Das, M., Wang, D., Yang, Z.: Modular checking for buffer overflows in the large. In: ICSE, pp. 232–241 (2006)
[24] Hoare, C.A.R.: Algorithm 63: Partition. Communications of the ACM 4(7), 321 (1961)
[25] Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's Doomed; We Can Prove It. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 338–353. Springer, Heidelberg (2009)
[26] Lindahl, T., Sagonas, K.F.: Practical type inference based on success typings. In: PPDP, pp. 167–178. ACM (2006)
[27] Logozzo, F., Ball, T.: Modular and verified automatic program repair. In: OOPSLA, pp. 133–146. ACM (2012)
[28] Meyer, B.: Eiffel: The Language. Prentice Hall (1991)
[29] Moy, Y.: Sufficient Preconditions for Modular Assertion Checking. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 188–202. Springer, Heidelberg (2008)
[30] Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The Yogi Project: Software Property Checking via Static Analysis and Testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
[31] Podelski, A., Rybalchenko, A., Wies, T.: Heap Assumptions on Demand. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 314–327. Springer, Heidelberg (2008)
[32] Popeea, C., Chin, W.-N.: Dual analysis for proving safety and finding bugs. In: SAC, pp. 2137–2143 (2010)
[33] Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61 (1995)
[34] Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, pp. 223–234 (2009)
[35] Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: ICSE (2011)
[36] Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: POPL, pp. 221–234 (2008)

Fixpoint Computation in the Polyhedra Abstract Domain Using Convex and Numerical Analysis Tools

Yassamine Seladji and Olivier Bouissou

CEA LIST
CEA Saclay Nano-INNOV Institut CARNOT
91 191 Gif sur Yvette CEDEX, France
firstname.lastname@cea.fr

Abstract. Polyhedra abstract domain is one of the most expressive and used abstract domains for the static analysis of programs. Together with Kleene algorithm, it computes precise yet costly program invariants. Widening operators speed up this computation and guarantee its termination, but they often induce a loss of precision, especially for numerical programs. In this article, we present a process to accelerate Kleene iteration with a good trade-off between precision and computation time. For that, we use two tools: convex analysis to express the convergence of convex sets using *support functions*, and numerical analysis to accelerate this convergence applying *sequence transformations*. We demonstrate the efficiency of our method on benchmarks.

1 Introduction

Static analysis plays an important role in software engineering. It allows to verify some safety properties on programs, like for example the absence of runtime errors, which is crucial in the verification of embedded programs. Static analysis uses the (concrete) program semantic function F to define a program invariant X, such that $F(X) = X$. This represents the set of reachable program states, but is however not computable. To deal with that, an over-approximation is introduced using abstract interpretation [10]. The main idea is to define a new (abstract) semantic function $F^\sharp$ that computes an abstract program invariant that includes the concrete one. This inclusion guarantees the safety of the result but not its accuracy: the larger the over-approximation is, the higher the probability to add false alarms. This over-approximation is due to the type of elements manipulated by $F^\sharp$; the most common and expressive abstraction is to represent the reachable states of numerical programs as convex polyhedra [9], that represent the linear relations existing between program variables.

The applicability of this analysis is faced with the compromise between precision and complexity: the standard algorithms (union, intersection) for computing with polyhedra are as precise as possible, but are mainly exponential in the number of program variables. A lot of work has been proposed to reduce this complexity, often at the cost of a loss in precision.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 149–168, 2013.

First, some less expressive domains were defined to decrease computational time, such as intervals, octagons [22] and templates [24]. These domains allow to express only a certain kind of linear relations between variables: for example, the octagon abstract domain encodes relations of the kind $\pm x \pm y \leq c$ for $c \in \mathbb{R}$. In the template abstract domain, the shape of the linear relations is fixed using a predefined matrix (the so-called template), and the efficiency of the analysis lies in the use of linear programming (LP) solvers that efficiently transform a general polyhedron into a template polyhedron. However, each of these domains limits the expressiveness of the computed invariants, which limits their use to analyse complex programs for which the shape of the relations between variables is difficult to establish a priori.

The other tool to reduce the computation time of the analysis is to modify Kleene algorithm to make it faster. The most classical way to do this is to use a widening operator [8] that ensures the termination of the analysis by extrapolating the limits of the iterates. The usual widening in the polyhedra domain removes constraints that are not stable from one iteration to the other. Even if it improves computation time, widening clearly creates a large over-approximation of the result. Some techniques were developed in complement of widening to minimize this loss of accuracy. For example, widening with thresholds [19] predefines a set of thresholds whose elements are used as limit candidates at each iteration. The difficulty lies in the choice of a relevant set of thresholds, which is often computed statically before the analysis. In our previous work [5], we presented a method that dynamically computes thresholds using numerical analysis tools. In [5], we defined this technique for the interval and octagon abstract domains, it can be smoothly extended to the template polyhedra domain. In this paper, we show how it can be extended to general polyhedra.

It can be noted that the solutions proposed to improve the computation time of a polyhedral analysis can be divided into two categories: on the one hand, a restricted representation of data to facilitate their manipulation, and on the other hand techniques that improve fixpoint computation, generally using a good widening operator. In this paper, we propose a novel method to accelerate the fixpoint computation in the polyhedral domain that uses both ideas together: polyhedra are represented using support functions [17,23] and Kleene algorithm is modified using sequence transformation methods [6]. The representation of polyhedra with support functions allows to use the notion of scalar convergence [25] of a sequence of polyhedra. We show that this convergence can be used with sequence transformation methods to accelerate the fixpoint computation with a good accuracy. Our main contribution is this method which offers a good balance between efficiency and precision, depending on our chosen polyhedra representation.

The paper is organized as follows: in Section 2, we present our method on a simple example, in Section 3, we introduce some useful definitions. In Section 4, we formally define our algorithm, and in Section 5, we compare it with the analysis using template polyhedra and present some techniques to improve the

Assume: $-20 \leq x \leq 20$
Assume:$0 \leq y \leq 3$
while 1 **do**
  if $y \geq 50 \wedge x \geq -20$ **then**
    y=x-0.1y+60
    x=0.2x+40
  else
    x=0.5x-y-2.5
    y=0.9y+10
  end if
end while

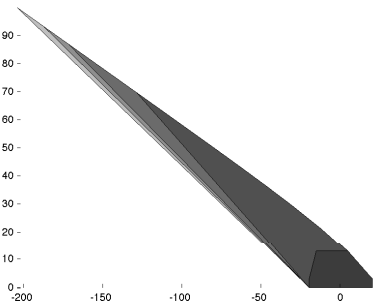


Fig. 1. The `prog` example, left, and some polyhedra obtained using Kleene iteration (iterates 1, 10, 20, 30 and 70, from dark to light), right

performance of our method. Some benchmarks are given in Section 6. Finally, we conclude with related work and some perspectives.

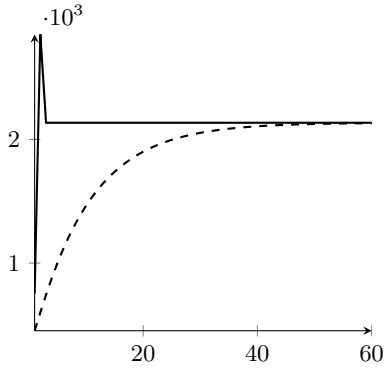
Notations. We define by $M_K(n, m)$ the set of matrices of a field K with n rows and m columns. Given $M \in M_K(n, m)$, $M_{i,j}$ is the element at the i^{th} row and j^{th} column. Given two vectors $v, w \in M_K(n, 1)$, let $\langle v, w \rangle \in K$ be the scalar product of v and w. For a set D, a sequence of elements in D will be noted $(x_k)_{k \in \mathbb{N}}$.

2 A Simple Example

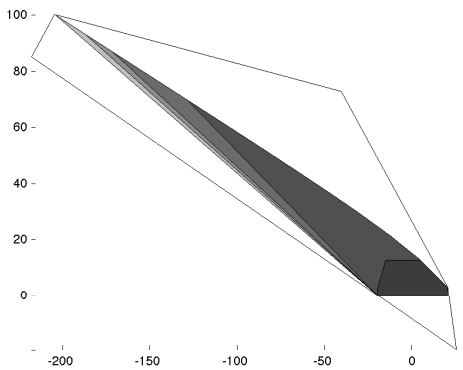
To illustrate our method, we consider the simple program presented in Figure 1 (left). It consists of a loop iterating a linear transformation, with a change if the variables enter the zone $x \geq -20$, $y \geq 50$. It can be shown that with the initial conditions $x \in [-20, 20]$ and $y \in [0, 3]$, the `then` branch of the loop is never taken. We used this condition to demonstrate the precision of our method compared to an analysis using template polyhedra.

A polyhedra based analysis of this program computes a sequence of polyhedra $(\mathbb{P}_k)_{k \in \mathbb{N}}$, some of which are depicted in Figure 1 (right). This sequence does not terminate, and widening does not help here: it discards the lower-left constraints thus computing an unbounded post-fixpoint while the least fixpoint is bounded. Note however that, in this case, the use of widening allows to prove that the program never enters the `then` branch but fails to compute a finite lower bound for variables x and y. To over-approximate the least fixpoint, our method works as follows. First, we chose a fixed set of directions Δ, i.e. a set of vectors in $\mathbb{R}^2$, and for each direction $d \in \Delta$, we extract the sequence of the values of the *support functions* of each polyhedron in this direction. As will be made clear in Section 3.2, the support function δ_P of a convex set P is a function such that $\forall d \in \mathbb{R}^2$, $\delta_P(d) = \sup\{\langle x, d \rangle : x \in P\}$. The property that we will use is that the sequence of support functions of the polyhedra $\mathbb{P}_k$ pointwise converges to the support function of the least fixpoint.

Now, let us study the values of the support functions in direction $X + 6Y$ (for example). We present in Figure 2(a) their values w.r.t. the number of iteration (dashed line). We see that this sequence slowly converges to its limit, and actually never reaches it. There exist many methods that help to quickly compute the limit of such numerical sequences. Among these methods we can use *sequence transformations* which automatically compute a new sequence that converges faster towards the same limit (even after finitely many steps in some cases). We depict in Figure 2(a) this new sequence (bold line); we clearly see that this sequence converges much faster towards its limit, after some irrelevant values due to the initialization of the process. Our method uses such a transformation to compute the limit l_d of the sequence of support functions in each direction $d \in \mathcal{\Delta}$. Once all the limits are computed, we construct a new polyhedron using the constraints $\langle x, d\rangle \leq l_d$ in each direction and insert it into Kleene iteration. This result is shown in Figure 2(b) and is a post-fixpoint (named $\mathbb{P}^\sharp$) of the program. Remark that this post-fixpoint is precise in the sense that the vertices of the least-fixpoint touch the faces of $\mathbb{P}^\sharp$. A more precise post-fixpoint can be computed using more directions, as will be shown in Section 6. Let us also note that this post-fixpoint is sufficiently precise to show that the `then` branch of the loop is never taken, while a template based analysis with the same directions could not prove that.



(a) The sequence of value of the support function in direction $X + 6Y$ (dashed) and the accelerated sequence (thick line).



(b) The polyhedra of Kleene algorithm (in dark) and the post-fixpoint computed using our method (in white).

Fig. 2. Results of our method on the example of Figure 1

3 Backgrounds

In this article, we are interested in computing invariants for programs with n variables using Kleene iteration on the polyhedra abstract domain. We denote the variables by $x_1, \ldots, x_n$ and we let $X = (x_1, \ldots, x_n)^t$ be the vector of all variables. We thus need to compute the limit of a sequence of convex subsets of $\mathbb{R}^n$.

We start by explaining how this sequence is defined, then we show that another notion of convergence (namely the scalar convergence) can be used to compute its limit. Finally, we present some methods that work well for accelerating the convergence of sequences of real values.

3.1 Polyhedra Abstract Domain

A convex polyhedron is a subset of $\mathbb{R}^n$ defined as the intersection of finitely many half-spaces, each half-space being given as a constraint of the form $\sum_{i=1}^{n} \alpha_i x_i \leq c$ where $\forall i \in [1, n],\ \alpha_i \in \mathbb{R}$ and $c \in \mathbb{R}$. We here adopt the constraint representation of polyhedron, which is best suited for static analysis. The set of all convex polyhedra has a lattice structure with an exact intersection operator and the convex hull as union. This was one of the first numerical abstract domains used in abstract interpretation [9]. Obviously, for implementation issues, the coefficients of the constraints are usually chosen as rational numbers and the set of constraints is encoded using a matrix representation.

Definition 1 (Polyhedra Abstract Domain). *The abstract domain of convex polyhedra is the set of all pairs (A, b) where $A \in M_{\mathbb{Q}}(k, n)$ is a matrix and $b \in \mathbb{Q}^k$ is a vector of dimension k for some $k \in \mathbb{N}$. A polyhedron $\mathbb{P}$ given by the pair (A, b) represents the set of all points $(x_1, \ldots, x_n) \in \mathbb{R}^n$ such that*

$$\forall j \in [1, k],\ \sum_{i=1}^{n} A_{j,i} x_i \leq b_j \ .$$

Let C^n be the abstract domain of convex polyhedra in $\mathbb{R}^n$.

Given a polyhedron $\mathbb{P} = (A, b)$, the i^{th} constraint of $\mathbb{P}$ is $\langle A_i, X \rangle \leq b_i$ where X is the vector of variables and A_i is the i^{th} line of A.

The static analysis of a program with the polyhedra domain consists in computing the least fixpoint $\mathbb{P}_\infty$ of a monotone map $F : C^n \to C^n$ given by the program semantics. To do so, the most used method is Kleene algorithm that uses the equation $\mathbb{P}_\infty = \bigsqcup_{k \in \mathbb{N}} F^k(\bot)$, where $\bot$ is the least element of C^n. Kleene iteration can be summarized by the following algorithm:

1: $\mathbb{P}_0 := \bot$
2: **repeat**
3: $\quad \mathbb{P}_i := \mathbb{P}_{i-1} \sqcup F(\mathbb{P}_{i-1})$
4: **until** $\mathbb{P}_i \sqsubseteq \mathbb{P}_{i-1}$

So we see that an abstract interpretation based static analysis of a program consists of defining a sequence of polyhedra $(\mathbb{P}_k)_{k \in \mathbb{N}}$ and computing its order-theoretic limit $\mathbb{P}_\infty = \bigsqcup_{k \in \mathbb{N}} \mathbb{P}_k$. As Kleene algorithm may not terminate, a widening operator is used to compute an over-approximation $\mathbb{P}_w \sqsupseteq \mathbb{P}_\infty$. In this article, we show that $\mathbb{P}_\infty$ can also be computed using scalar convergence and that numerical acceleration methods can be used to quickly compute an approximation of the scalar limit.

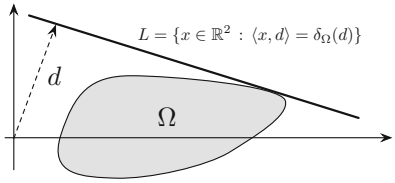


Fig. 3. Geometrical interpretation of the support function: the line L represents the limit of the half-space $H = \left\{x \in \mathbb{R}^2 : \langle x, d\rangle \leq \delta_\Omega(d)\right\}$, in which Ω lies

3.2 Convex Analysis Tools

A convenient representation of a convex set is the use of its support function [17,23]. In this section, we define the notion of support function and present how they can be computed in the case of convex polyhedron. We also introduce the notion of scalar convergence.

Definition 1 (Support Function). *Let $\Omega \subseteq \mathbb{R}^n$ be a non-empty convex set. The support function $\delta_\Omega : \mathbb{R}^n \mapsto \mathbb{R} \cup \{+\infty\}$ of Ω is given by*

$$\forall d \in \mathbb{R}^n,\ \delta_\Omega(d) = \sup\{\langle x, d\rangle : x \in \Omega\}\ .$$

As stated by Definition 1, the support function is defined only for the non-empty convex set. In the following, we consider only non-empty convex sets. The support function of a convex set Ω associates to each *direction* $d \in \mathbb{R}^n$ the biggest value of the scalar product $\langle x, d\rangle$ for $x \in \Omega$. The interest of this notion is that any convex set Ω is uniquely and completely determined by the values of its support function for all $d \in \mathbb{R}^n$ (see Property 1). So the set of all support functions (i.e. the set of positive homogeneous, convex real valued functions over $\mathbb{R}^n$) is isomorphic to the set of convex sets over $\mathbb{R}^n$.

Property 1. *Let Ω be a non-empty convex set and δ_Ω be its support function. We have:*

$$\Omega = \bigcap_{d \in \mathbb{R}^n} \{x \in \mathbb{R}^n : \langle x, d\rangle \leq \delta_\Omega(d)\}\ .$$

The value of δ_Ω in a direction $d \in \mathbb{R}^n$ might be infinite depending on whether Ω is bounded in this direction. Figure 3 gives a geometrical interpretation of δ_Ω: in the direction d, $\delta_\Omega(d)$ defines the smallest half-space (delimited by the thick line L) that contains Ω. When Ω is a polyhedron represented by its constraints system, its support function can be obtained using *linear programming*.

Let now $(\Omega_k)_{k\in\mathbb{N}}$ be a sequence of convex sets, with $\Omega_k \subseteq \mathbb{R}^n$ for all $k \in \mathbb{N}$. From a topological point of view, there are many ways to define the convergence of the sequence $(\Omega_k)_{k\in\mathbb{N}}$ [21], each of them leading to a (possibly different) limit. We choose to use the notion of scalar-convergence because it is based on support function and because it corresponds to the notion of limit used in abstract interpretation when the sequence $(\Omega_k)_{k\in\mathbb{N}}$ is monotone.

Definition 2 (Scalar-convergence). *Let $(\Omega_k)_{k\in\mathbb{N}}$ be a sequence of convex sets. For each $k \in \mathbb{N}$, let δ_{Ω_k} be the support function of Ω_k. Let Ω be a convex set and let δ_Ω be its support function. We say that $(\Omega_k)_{k\in\mathbb{N}}$ scalar-converges (or S-converges) to Ω, denoted by* $\text{s-lim}_{k\to+\infty}\,\Omega_k = \Omega$, *iff*

$$\forall d \in \mathbb{R}^n,\ \lim_{k\to+\infty}(\delta_{\Omega_k}(d)) = \delta_\Omega(d)\ .$$

The S-convergence defines the limit of a sequence of convex sets $(\Omega_k)_{k\in\mathbb{N}}$ via infinitely many limits of numerical sequences $(\delta_{\Omega_k}(d))_{k\in\mathbb{N}}$, for all $d \in \mathbb{R}^n$. Property 2 shows that the S-convergence of an increasing sequence is the supremum of its elements.

Property 2. *Let $(\Omega_k)_{k\in\mathbb{N}}$ be a sequence of closed convex sets and let cl be the convex closure function. If we have that $\forall k \in \mathbb{N},\ \Omega_k \subseteq \Omega_{k+1}$, then*

$$\text{s-lim}_{k\to\infty}\,\Omega_k = cl(\bigcup_{k\geq 0}\Omega_k)\ .$$

Recall that, as defined in Section 3.1, Kleene algorithm computes program invariants as the union of a sequence of convex polyhedra. These polyhedra form an increasing sequence, so Property 2 shows that the S-convergence can be used to compute the result of Kleene iteration. We use this idea in Section 4 to define an accelerated version of Kleene algorithm for convex polyhedra.

3.3 Numerical Analysis Tools

In this section, we present techniques called sequence transformations that are used to quickly compute the limit of a numerical sequence of real numbers. These techniques were already used in [5] to accelerate the convergence of the fixpoint computation for the box or octagon abstract domains. We recall the basic notions of sequence transformation that are needed to understand our framework, a good review on the theory and applications of these methods can be found in [6].

We equip $\mathbb{R}$ with the euclidean distance and define $\mathbf{Seq}(\mathbb{R})$ as the set of all sequences of real numbers (i.e. functions from $\mathbb{N}$ to $\mathbb{R}$). We say that a sequence $(x_k)_{k\in\mathbb{N}} \in \mathbf{Seq}(\mathbb{R})$ converges towards $x \in \mathbb{R}$, denoted by $\lim_{k\to\infty} x_k = x$, if $\lim_{k\to\infty} |x_k - x| = 0$. More formally, we have:

$$\lim_{k\to\infty} x_k = x \Leftrightarrow \forall\varepsilon > 0,\ \exists K \in \mathbb{N} : \forall k \geq K,\ |x_k - x| \leq \varepsilon\ .$$

Given two sequences $(x_k)_{k\in\mathbb{N}}$ and $(y_k)_{k\in\mathbb{N}}$ with the same limit ℓ, we say that (y_k) converges *faster* to ℓ than (x_k) if $\lim_{k\to\infty}\left(\frac{y_k-l}{x_k-l}\right) = 0$. The goal of sequence transformations is to automatically compute, from a slowly converging sequence (x_k), a sequence (y_k) that converges towards the same limit faster than (x_k). In this way, we can use (y_k) to quickly obtain an approximation of the limit of (x_k). This is formally stated in Definition 3.

Definition 3. *A* sequence transformation *is a function* $T : \mathbf{Seq}(\mathbb{R}) \to \mathbf{Seq}(\mathbb{R})$ *such that, for all* converging *sequences* $(x_k)_{k\in\mathbb{N}} \in \mathbf{Seq}(\mathbb{R})$*, the sequence* (y_k) *defined by* $(y_k) = T(x_k)$ *is convergent with* $\lim_{k\to\infty} y_k = \lim_{k\to\infty} x_k$*. The sequence* (y_k) *is said to be* accelerated *if* $\lim_{k\to\infty} \frac{y_k - l}{x_k - l} = 0$*.*

An example of a sequence transformation is the well-known Δ^2-Aitken transformation $\Theta : \mathbf{Seq}(\mathbb{R}) \to \mathbf{Seq}(\mathbb{R})$, defined by:

$$\forall \mathbf{x} \in \mathbf{Seq}(\mathbb{R}),\ \forall k \in \mathbb{N},\ \Theta(\mathbf{x})_k = x_k - \frac{(x_{k+1} - x_k)^2}{x_k - 2x_{k+1} + x_{k+2}}\ .$$

We apply this transformation method to $\mathbf{x}_k = 1 + \frac{1}{k+1}$ $(\forall k \in \mathbb{N})$, a sequence that converges to 1. The result is given in the following table, where $(\Theta(\mathbf{x})_k)_{k\in\mathbb{N}}$ converges faster than $(\mathbf{x}_k)_{k\in\mathbb{N}}$ toward 1.

$\mathbf{x}_k$	2.00	1.5	1.33	1.25	1.2	1.16	1.14	1.125	1.11
$\Theta(\mathbf{x})_k$		1.25	1.16	1.12	1.10	1.08	1.07	1.06	

For more details on Δ^2-Aitken transformation see [7].

Obviously, a sequence transformation does not accelerate all converging sequences, i.e. $(T(x_k))_{k\in\mathbb{N}}$ does not always converge faster than $(x_k)_{k\in\mathbb{N}}$. Remark however that it is required that $T(x_k)$ still converges towards the same limit. An important notion is the kernel of a sequence transformation, which is the set of all sequences $\mathbf{x} \in \mathbf{Seq}(\mathbb{R})$ such that $T(\mathbf{x})$ is ultimately constant, see Definition 4.

Definition 4. *Let* $T : \mathbf{Seq}(\mathbb{R}) \to \mathbf{Seq}(\mathbb{R})$ *be a sequence transformation. The* kernel *of* T*, denoted by* $K(T) \subseteq \mathbf{Seq}(\mathbb{R})$*, is the set of sequences defined by:*

$$\forall \mathbf{x} \in \mathbf{Seq}(\mathbb{R}),\ \mathbf{x} \in K(T) \Leftrightarrow \mathbf{y} = T(\mathbf{x})\ \textit{and}\ \exists n \in \mathbb{N} : \forall k \geq n,\ y_k = y_n\ .$$

So clearly, sequences in the kernel of a transformation T are very interesting because we can compute their limit in a finite time by looking at the elements of the accelerated sequence. However, computing exactly the kernel of a sequence transformation is very complicated [7]. In our experimentations, we used the ϵ-algorithm [26], which is often cited as the best general purpose acceleration algorithm. The ϵ-algorithm is a generalization of the Δ^2-algorithm that is less sensible to numerical instability. For the sake of conciseness, we do not present it, let us just mention that its kernel contains a large class of sequences like convergent linear sequences and *totally monotonic* ones [6, Chap. 2, pp. 85–91].

4 The Acceleration Process

In this section, we present our main contribution which is a new fixpoint algorithm on the polyhedra abstract domain. For that, we define in Section 4.1 an accelerated version of the S-convergence, called *the accelerated S-convergence*. In section 4.2, we propose an abstract version of the accelerated S-convergence which is used in Section 4.3 for the fixpoint computation.

4.1 The Accelerated S-Convergence

In this section, we show how support function can be combined with sequence transformations to accelerate the s-convergence of convex polyhedra sequences. The method we develop is called the *accelerated S-convergence*.

Now, let Ω be a convex set and $(\mathbb{P}_k)_{k\in\mathbb{N}}$ be a sequence of polyhedra such that $\text{s-lim}_{k\to+\infty}\mathbb{P}_k = \Omega$. We want to compute Ω in a fast way. Let $\delta_{\mathbb{P}_k}$ be the support functions of $\mathbb{P}_k$ for all $k \in \mathbb{N}$. We put $\forall d \in \mathbb{R}^n,\ \forall k \in \mathbb{N},\ S_k^d = \delta_{\mathbb{P}_k}(d)$. From Definition 2, we have that if $\lim_{k\to+\infty} S_k^d = S_d$, then $S_d = \delta_\Omega(d)$. It means that $\Omega = \bigcap_{d\in\mathbb{R}^n}\{x \in \mathbb{R}^n : \langle x, d\rangle \leq S_d\}$. So, the S-limit of $(\mathbb{P}_k)_{k\in\mathbb{N}}$ can be defined using the limit of numerical sequences $(S_k^d)_{k\in\mathbb{N}}$, for all $d \in \mathbb{R}^n$.

Property 3. *Let $(\mathbb{P}_k)_{k\in\mathbb{N}}$ be a convex polyhedra sequence, and $\delta_{\mathbb{P}_k}$ be the support function of each $\mathbb{P}_k$.*

$$\textit{If } (\forall d \in \mathbb{R}^n), \lim_{k\to+\infty}\delta_{\mathbb{P}_k}(d) = S_d \textit{ then } \text{s-lim}_{k\to+\infty}\mathbb{P}_k = \bigcap_{d\in\mathbb{R}^n} H_d$$

where $H_d = \{x \in \mathbb{R}^n : \langle x, d\rangle \leq S_d\}$ is a supporting hyperplane of the S-limit of $(\mathbb{P}_k)_{k\in\mathbb{N}}$.

Property 3 shows that it is possible to use numerical methods of Section 3.3 to accelerate the computation of the S-limit of $(\mathbb{P}_k)_{k\in\mathbb{N}}$. Let T be a sequence transformation as presented in Section 3.3. We compute the sequence $(T(S_k^d))_{k\in\mathbb{N}}$ for all $d \in \mathbb{R}^n$, we assume that the sequence $(S_k^d)_{k\in\mathbb{N}}$ belongs to the kernel of T (see Definition 4). So $(T(S_k^d))_{k\in\mathbb{N}}$ converges faster than $(S_k^d)_{k\in\mathbb{N}}$ towards S_d, thus accelerating the computation of Ω. This is stated by Definition 5.

Definition 5 (Accelerated S-convergence). *Let $(\mathbb{P}_k)_{k\in\mathbb{N}}$ be a convex polyhedra sequence. For each $k \in \mathbb{N}$, let $\delta_{\mathbb{P}_k}$ be the support function of $\mathbb{P}_k$, and let T be a sequence transformation. The accelerated S-convergence, noted* $\text{s}_\text{A}\text{-lim}$*, is defined as:*

$$\text{s}_\text{A}\text{-lim}_{k\to\infty}\mathbb{P}_k = \bigcap_{d\in\mathbb{R}^n}\left\{x \in \mathbb{R}^n : \langle x, d\rangle \leq \lim_{k\to+\infty} T(\delta_{\mathbb{P}_k}(d))\right\} .$$

In particular, we have $\text{s-lim}_{k\to+\infty}\mathbb{P}_k = \text{s}_\text{A}\text{-lim}_{k\to\infty}\mathbb{P}_k$.

In practice, the s_A-lim of $(\mathbb{P}_k)_{k\in\mathbb{N}}$ cannot be used because we must compute $\lim_{k\to+\infty} T(\delta_{\mathbb{P}_k}(d))$ for all d in $\mathbb{R}^n$. We can easily prove that this set can be restricted to directions in the unit ball $\mathbb{B}^n$, but then the accelerated S-limit still required infinitely many limit computations. In Section 4.2, a finite abstraction of the accelerated S-convergence, called *the abstract S-convergence*, is defined.

4.2 The Abstract S-Convergence

Let Ω be a convex set, and $\mathbb{B}^n \subseteq \mathbb{R}^n$ be the unit ball. Using support function properties, we have that $\forall d \in \mathbb{B}^n,\ \Omega \subseteq H_d$ with $H_d = \{x \in \mathbb{R}^n : \langle x, d\rangle \leq \delta_\Omega(d)\}$. In particular, Ω is included in every intersection of finitely many supporting

hyperplanes. So we can over-approximate Ω using a finite set of directions and computing each supporting hyperplane in these directions. Property 4 presents that.

Property 4. *For a convex set Ω, we have :*

- $(\forall \Delta \subseteq \mathbb{B}^n), \Omega \subseteq \bigcap_{d_i \in \Delta} \{x \in \mathbb{R}^n : \langle x, d_i \rangle \leq \delta_\Omega(d_i)\}$.
- *If* $(\Delta = \mathbb{B}^n)$ *then* $\Omega = \bigcap_{d_i \in \Delta} \{x \in \mathbb{R}^n : \langle x, d_i \rangle \leq \delta_\Omega(d_i)\}$.

In the sequel, we define the set $\Lambda = \mathcal{P}(\mathbb{B}^n)$, Λ is the power set of $\mathbb{B}^n$. $(\Lambda, \subseteq_\Lambda)$ forms a complete lattice with $\bot = \emptyset$, $\top = \mathbb{B}^n$, $\subseteq_\Lambda, \cup_\Lambda$ and $\cap_\Lambda$ being the usual set operations.

The abstract S-convergence applies the accelerated S-convergence on an element Δ of the lattice $(\Lambda, \subseteq_\Lambda)$ to compute an over-approximation of the limit of $(\mathbb{P}_k)_{k \in \mathbb{N}}$. The idea is to apply the accelerated S-convergence partially using directions in Δ. This is defined in Definition 6.

Definition 6 (The Abstract S-convergence). *Let $(\Lambda, \subseteq_\Lambda)$ be the lattice of direction sets. The abstract S-convergence of a sequence $(\mathbb{P}_k)_{k \in \mathbb{N}} \subseteq C^n$, noted $\mathrm{s}^\sharp_\mathrm{A}$-lim, is a function from $\Lambda \times (\mathbb{N} \to C^n)$ to C^n, such that:*

$$\forall \Delta \in \Lambda,\ \mathrm{s}^\sharp_\mathrm{A}\text{-lim}(\Delta, \mathbb{P}_k) = \bigcap_{d_i \in \Delta} \left\{x \in \mathbb{R}^n : \langle x, d_i \rangle \leq \lim_{k \to +\infty} T(\delta_{\mathbb{P}_k}(d_i))\right\}$$

where T is a sequence transformation and $\delta_{\mathbb{P}_k}$ are the support functions of $\mathbb{P}_k$.

Now, we can consider *the abstract S-convergence* as a finite approximation of the accelerated S-convergence, if the chosen direction set Δ is finite. As stated by Property 4, it computes an over-approximation of the S-limit of a polyhedra sequence.

Property 5. *For a sequence $(\mathbb{P}_k)_{k \in \mathbb{N}} \subseteq C^n$, we have that, if $\mathrm{s}_\mathrm{A}\text{-lim}_{k \to +\infty}(\mathbb{P}_k) = \Omega$ then:*

- $(\forall \Delta \in \Lambda), \Omega \subseteq \mathrm{s}^\sharp_\mathrm{A}\text{-lim}(\Delta, \mathbb{P}_k) = \Omega^\sharp$, *where $\Omega^\sharp$ is the best abstraction of Ω using the direction set Δ, i.e.* $\Omega^\sharp = \bigcap_{d_i \in \Delta} \left\{x \in \mathbb{R}^n : \langle x, d_i \rangle \leq \delta_\Omega(d_i)\right\}$.
- $(\forall \Delta_1, \Delta_2 \in \Lambda)$, *if* $\Delta_1 \subseteq_\Lambda \Delta_2$ *then* $\Omega \subseteq \mathrm{s}^\sharp_\mathrm{A}\text{-lim}(\Delta_2, \mathbb{P}_k) \subseteq \mathrm{s}^\sharp_\mathrm{A}\text{-lim}(\Delta_1, \mathbb{P}_k)$

Informally Property 5 says that the more directions we have, the more precise the result will be. In the case where Ω is a polyhedron, there exists a minimal set $\Delta_\Omega \subseteq \Lambda$, such that $\Omega = \cap_{d \in \Delta_\Omega} \{x \in \mathbb{R}^n : \langle x, d \rangle \leq \delta_\Omega(d)\}$. This Δ_Ω represents the set of all normal vectors of the constraints of Ω. However, these constraints are generally unknown, so we do not know Δ_Ω. Even worse, when Ω is not a polyhedron, there is no finite set Δ which is optimal to compute $\mathrm{s}^\sharp_\mathrm{A}\text{-lim}(\Delta, \mathbb{P}_k)$ (because there is no best abstraction of a general convex set into the abstract domain of polyhedra). The efficiency and precision of our method depends on the choice of a relevant set Δ. We discuss this choice in Section 5.2.

Algorithm 1. Accelerated Kleene Algorithm.

Input: $\Delta \in \Lambda$, finite
Input: $\varepsilon > 0$
1: $\mathbb{P}_0 := \bot$
2: $i := 1$
3: $\mathbb{P}^\sharp := \{\}$ *//Initialize $\mathbb{P}^\sharp$ to the empty sequence*
4: **repeat**
5: $\quad \mathbb{P}_i := \mathbb{P}_{i-1} \sqcup F^\sharp(\mathbb{P}_{i-1})$
6: $\quad \mathbb{P}^\sharp = \mathbb{P}^\sharp \cup \{\mathbb{P}_i\}$ *//Add the result of the iteration $\mathbb{P}_i$ to $\mathbb{P}^\sharp$*
7: $\quad \mathbb{P}^\sharp_{\infty(i)} := \mathrm{s}^\sharp_{\mathrm{A}}\text{-lim}\left(\Delta, \mathbb{P}^\sharp\right)$
8: $\quad$ **if** $(\text{distance}(\mathbb{P}^\sharp_{\infty(i)}, \mathbb{P}^\sharp_{\infty(i-1)}) \leq \varepsilon)$ **then**
9: $\quad\quad \mathbb{P}_i := \mathbb{P}^\sharp_{\infty(i)}$
10: $\quad$ **end if**
11: $\quad i := i + 1$
12: **until** $\mathbb{P}_i \sqsubseteq \mathbb{P}_{i-1}$

4.3 The Accelerated Kleene Iteration Algorithm

In this section, we use *the abstract S-convergence* with Kleene iteration to accelerate fixpoint computation. This improvement proposes a trade-off between precision and computation time by including more directions in the set Δ used for the abstract S-convergence. If we run the Kleene algorithm with the polyhedra abstract domain, the collection of successive iterates forms a sequence of convex polyhedra, noted $(\mathbb{P}_k)_{k\in\mathbb{N}}$, such that:

$$\begin{cases} \mathbb{P}_0 & = \bot \\ \mathbb{P}_{k+1} & = \mathbb{P}_k \sqcup F^\sharp(\mathbb{P}_k), (\forall k \in \mathbb{N}) \end{cases}$$

We assume that $\forall k \geq 1, \mathbb{P}_k \neq \bot$.

The abstract semantic function $F^\sharp$ is monotone by definition and $(\mathbb{P}_k)_{k\in\mathbb{N}}$ is an increasing sequence, i.e. $\forall k \in \mathbb{N},\ \mathbb{P}_k \sqsubseteq \mathbb{P}_{k+1}$. As stated in Property 2, the S-limit of an increasing sequence is the convex hull of its elements, so for the sequence of Kleene iterates, the S-limit of $(\mathbb{P}_k)_{k\in\mathbb{N}}$ is the least fixpoint of $F^\sharp$, denoted $\mathbb{P}_\infty$. So we have:

$$\begin{aligned} & \mathbb{P}_\infty = \ \text{s-lim}_{k\to+\infty}\, \mathbb{P}_k \\ \text{So} \quad & \mathbb{P}_\infty = \mathrm{s}_{\mathrm{A}}\text{-lim}_{k\to+\infty}\, \mathbb{P}_k \quad \text{(By transitivity).} \\ \text{Thus } (\forall \Delta \in \Lambda), & \mathbb{P}_\infty \sqsubseteq \ \mathrm{s}^\sharp_{\mathrm{A}}\text{-lim}(\Delta, \mathbb{P}_k) \quad \text{(Using Property 5).} \end{aligned}$$

This shows that we can compute an over-approximation of $\mathbb{P}_\infty$ using the new notion of convergence introduced in Section 4.2. Note that the quality of the over-approximation depends on the choice of the direction set Δ.

In Algorithm 1, we define a new accelerated Kleene algorithm, which is the standard Kleene algorithm combined with the abstract S-convergence. The main idea is to compute in parallel the sequence $(\mathbb{P}_k)_{k\in\mathbb{N}}$ and its $\mathrm{s}^\sharp_{\mathrm{A}}$-lim. Once this limit is computed, we use it as a fixpoint candidate.

Using a direction set Δ given as an input, in each iteration Algorithm 1 computes the abstract element $\mathbb{P}_i$ and puts it as a new element of $\mathbb{P}^\sharp$. We have that $\mathbb{P}^\sharp_{\infty(i)}$ is the result of the abstract S-convergence applied on $\mathbb{P}^\sharp$ and Δ. Collecting these results, we obtain the accelerated sequence, called $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$. So we construct simultaneously $\mathbb{P}^\sharp$ and $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$. When the algorithm detects that the sequence $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$ stabilizes, we assume that it is close to its limit and thus we use the last element of $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$, noted $\mathbb{P}^\sharp_{\infty(i)}$, as a relevant threshold, i.e. we modify the current Kleene iterate $\mathbb{P}_i$ to be $\mathbb{P}^\sharp_{\infty(i)}$. Thanks to the properties of the abstract S-convergence, this threshold is obtained after a few iterations, and it is a good approximation of the fixpoint.

Algorithm 1 detects the stabilization of the sequence $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$ by computing the distance between two successive elements. The distance d_Δ we use is given by:

$$\forall \mathbb{P}_1, \mathbb{P}_2 \in C^n,\ d_\Delta(\mathbb{P}_1, \mathbb{P}_2) = \sup_{d_i \in \Delta} |\delta_{\mathbb{P}_1}(d_i) - \delta_{\mathbb{P}_2}(d_i)| \ .$$

Clearly if Δ is finite (or $\Delta \subset \mathbb{B}^n$), d_Δ is not a distance on C^n. In particular, there exist infinitely many pairs $(\mathbb{P}, \mathbb{P}')$, $\mathbb{P} \neq \mathbb{P}'$, with $d_\Delta(\mathbb{P}, \mathbb{P}') = 0$. However, the sequence $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$ is made of polyhedra whose directions are given by the set Δ, i.e. they are template polyhedra. The function d_Δ is a distance between template polyhedra and can be used to detect the stabilization of the sequence. Moreover, it can be computed in linear time as the support function in a direction d_i are just the inhomogeneous term of the associated constraint. So we say that the sequence $(\mathbb{P}^\sharp_{\infty(k)})_{k\in\mathbb{N}}$ has stabilized when $d_\Delta(\mathbb{P}^\sharp_{\infty(i)}, \mathbb{P}^\sharp_{\infty(i-1)}) \leq \varepsilon$, where $\varepsilon > 0$ is a user-defined parameter (usually around 10^{-3}).

We can prove that Algorithm 1 terminates when $\mathbb{P}^\sharp$ can be accelerated by $\mathrm{s}^\sharp_{\mathrm{A}}$-lim, i.e. when $\mathbb{P}^\sharp$ belongs to the kernel of the sequence transformation used to compute $\mathrm{s}^\sharp_{\mathrm{A}}$-lim. Note that practical experiments show that many sequences are accelerated even if they are not in the kernel, so we believe that Algorithm 1 can be efficiently used for many types of programs. However, it's hard to establish a priori if a sequence will be accelerated, and we know that no method accelerates all convergent sequences [12]. To overcome this, we combine our method with widening by replacing lines 6 to 10 of Algorithm 1 by:

if $i \leq nbDelay$ **then**
  $\mathbb{P}^\sharp = \mathbb{P}^\sharp \cup \{\mathbb{P}_i\}$
  $\mathbb{P}^\sharp_{\infty(i)} := \mathrm{s}^\sharp_{\mathrm{A}}\text{-lim}\left(\Delta, \mathbb{P}^\sharp\right)$
  if (distance$(\mathbb{P}^\sharp_{\infty(i)}, \mathbb{P}^\sharp_{\infty(i-1)}) \leq \varepsilon$) **then**
    $\mathbb{P}_i := \mathbb{P}^\sharp_{\infty(i)}$
  end if
else
  $\mathbb{P}_i := \mathbb{P}_{i-1} \nabla \mathbb{P}_i$ // *The widening operator.*
end if

The idea is similar to widening with delay: we apply the accelerated S-convergence during the first $nbDelay$ iterations. If the computation doesn't terminate we use the widening to force the termination. In our experiments, however, we did not need to use the widening to make the analysis terminate. Note that in the case where $s^\sharp_A$-lim accelerates sequences obtained in some directions and not the others, a way to improve this algorithm is to use the polyhedron defined by the accelerated directions as a threshold for widening. This allows to keep the information obtained by $s^\sharp_A$-lim and thus improves the precision of the widening result. Note that other widening techniques, as defined in [2], can also be smoothly combined with our technique.

5 Performance

The performance of the method presented in this paper mainly depends on two parameters. First, the choice of the transformation method to accelerate the sequences convergence is important. Clearly, each transformation accelerates very well sequences in its kernel (see definition in Section 3.3), so we must choose an algorithm with a large kernel. In our experimentations, we used the ε-algorithm. Second and mainly, our method depends on the choice of directions used to compute the abstract S-limit. We discuss this choice in Section 5.2. The direction set used in our technique can be seen as a template defined in [24]. We next emphasize this comparison and the differences between both methods.

5.1 Comparison with Template Abstract Domain

The template abstract domain [24] represents a derivation of the polyhedra domain in which the polyhedra are restricted to have a fixed shape. This shape is defined by a $n \times m$ matrix, called template constraint matrix (TCM), where n is the number of program variables and m the number of constraints used to define the polyhedra shape. The analogue of the TCM in our method is the direction set Δ: each line of a TCM is a direction, so clearly the set of all TCMs is equivalent to the lattice Λ of direction sets. Given a TCM Γ, we denote by T_Γ the template domain with template Γ and by $\Delta_\Gamma \in \Lambda$ the equivalent direction set; we want to compare the fixpoint we obtain using Kleene iteration in the T_Γ and $s^\sharp_A$-lim$(\Delta_\Gamma, \mathbb{P}_k)$.

Let Γ be a TCM, and $\alpha_\Gamma : C^n \to T_\Gamma$ be the abstraction function for the template domain, such that $\forall \mathbb{P} \in C^n$ represented as a conjunction of constraints of the form (A,b) (see Definition 1), $\alpha_\Gamma(\mathbb{P}) = \cap_{d_i \in \Gamma}\{x |\, \langle x, d_i\rangle \leq c_i\}$, where c_i is the solution of the following problem: min $b^T\lambda$ s.t $\lambda \geq 0 \wedge A^T\lambda = d_i$. Note that $c_i = \delta_\mathbb{P}(d_i)$, so $\alpha_\Gamma(\mathbb{P})$ can be defined as:

$$\alpha_\Gamma(\mathbb{P}) = \cap_{d_i \in \Gamma}\{x |\, \langle x, d_i\rangle \leq \delta_\mathbb{P}(d_i)\} \ .$$

Let now P be a program and $F^\sharp$ be its abstract semantic function in the polyhedra domain. An analysis of P in the template domain computes the invariant $\mathbb{P}^t_\infty = \sqcup^t \mathbb{P}^t_k$ such that:

$$\begin{cases} \mathbb{P}^t_0 & = \alpha_\Gamma(\mathbb{P}_0) \\ \mathbb{P}^t_{k+1} & = \mathbb{P}^t_k \sqcup^t \alpha_\Gamma(F^\sharp(\mathbb{P}_k)),\ \forall k \in \mathbb{N} \end{cases} .$$

In other words, it performs the standard Kleene iteration, but abstracts each intermediate value into the template abstract domain. Here $\sqcup^t$ is the union in T_Γ which is very easy to compute [24].

Let $(\mathbb{P}_k)_{k\in\mathbb{N}}$ be the sequence computed by Kleene iteration in the polyhedra domain. It is easy to prove by induction that $\forall k \in \mathbb{N},\ \alpha_\Gamma(\mathbb{P}_k) \sqsubseteq \mathbb{P}^t_k$. So $\sqcup^t_{k\in\mathbb{N}} \alpha_\Gamma(\mathbb{P}_k) \sqsubseteq \sqcup_{k\in\mathbb{N}} \mathbb{P}^t_k = \mathbb{P}^t_\infty$. As α_Γ is continuous, we know that:

$$\alpha_\Gamma\big(\sqcup_{k\in\mathbb{N}}\, \mathbb{P}_k\big) \sqsubseteq \mathbb{P}^t_\infty. \tag{1}$$

Let $\mathbb{P}_\infty = \sqcup_{k\in\mathbb{N}} \mathbb{P}_k$ be the least fixpoint of $F^\sharp$ in the polyhedra domain. From the definitions of α_Γ and $\mathrm{s}^\sharp_\mathrm{A}$-lim$(\Delta_\Gamma, \mathbb{P}_k)$, we can easily prove that:

$$\mathrm{s}^\sharp_\mathrm{A}\text{-lim}(\Delta_\Gamma, \mathbb{P}_k) = \alpha_\Gamma(\mathbb{P}_\infty)\ . \tag{2}$$

From Equation 1 and 2, we obtain that $\mathrm{s}^\sharp_\mathrm{A}$-lim$(\Delta_\Gamma, \mathbb{P}_k) \sqsubseteq \mathbb{P}^t_\infty$.

It means that, using the same TCM, the result of our method is more precise than the one obtained with template abstract domain. The cause is that, in the template case, all the analysis is made in a less expressive domain, so some over-approximation is added at each iteration. In our method, the over-approximation is done once, when the result of the abstract S-convergence is injected in the Kleene iteration to accelerate its termination. From Equation 2, our method automatically computes the best abstraction of the fixpoint in the template domain. The use of numerical acceleration methods allows to compute it without having to compute the fixpoint itself.

5.2 Discussion on Direction Set

Given a direction set Δ, the abstract S-convergence computes the template abstraction of the least fixpoint computed by Kleene iterates. So clearly, the choice of Δ has a major influence on the quality of the abstraction. Moreover, as we want to stabilize every sequence $\delta_{\mathbb{P}_k}(d)$ for all $d \in \Delta$, the choice of Δ also influences the performance of the algorithm as some sequences will be less likely to be accelerated. However, there is no best direction set and choosing a good one is a difficult problem [24]. We mainly have two methods to choose the directions.

Uniformly Distributed Directions. First, one can choose directions that are uniformly distributed on a surface of the n-dimensional sphere (n represents the program dimension). This technique guarantees that the entire space is covered, so that if the limit of Kleene iterates is bounded, our method computes a bounded polyhedra as well. This technique is also used in [14], where support functions are used to represent convex sets for hybrid systems analysis. This technique however does not consider the dynamics of the program to choose the directions and is thus often not optimal.

Using Partial Traces. A better solution is to use and extend the work of [1] where statistical tools, namely *principal component analysis* (or PCA), are used to refine the box abstract domain by changing the axis in which boxes are defined. PCA is a commonly used tool in statistics for analysing data. Given a set of points in a n-dimensional space, PCA computes an orthogonal coordinate system that better represents the points, i.e. the new axes maximize the variance of the projection of the initial values. Then, we can remove the most extreme points and we obtain new axes, that are not orthogonal to the first, and that best represent the reduced set of points. Iterating this idea, we obtain a set of directions that contain many information on how the points are distributed in space.

In our case, we generate points by computing partial evaluation traces of the program and collecting all the values of the variables. After the PCA analysis, we obtain a direction set Δ with which we can perform our static analysis as in Algorithm 1. We present in Section 6 the results we obtain using this technique, which are very encouraging and our future work will consist in applying this PCA analysis dynamically to discover relevant directions.

5.3 Case of Affine Programs

The main bottleneck of our algorithm is the fact that we must compute the polyhedra given by Kleene iteration before computing the support functions in the chosen directions. For programs with many variables, this quickly becomes impossible as Kleene iteration is very time consuming. When the program iterates an affine transformation (i.e. when the semantic function F is of the form $F(X) = AX + b$, with A a matrix and b a vector), we can overcome this problem by directly computing the value of the support function of $\mathbb{P}_i$ in each direction without computing $\mathbb{P}_i$, using ideas from [4]. We briefly describe this method here. On the one hand, using Kleene algorithm and the semantic function F, $\mathbb{P}_i$ is obtained by :

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup F(\mathbb{P}_{i-1}). \tag{3}$$

On the other hand, the support functions of convex sets can be computed efficiently using the following operations. For any convex sets $S, S' \subseteq C^n$, we have:

- $\forall M \in M_{\mathbb{R}}(n,m),\ \delta_{MS}(d) = \delta_S(M^T d)$.
- $\delta_{S \sqcup S'}(d) = \max(\delta_S(d), \delta_{S'}(d))$.
- $\delta_{S \oplus S'}(d) = \delta_S(d) + \delta_{S'}(d)$.

In these formula, MS denotes the transformation of S by M, such that $MS = \{Mx | x \in S\}$ and $S \oplus S'$ denotes the Minkowski sum: $S \oplus S' = \{x + x' \mid x \in S,\ x' \in S'\}$. Using these properties and Equation 3, the support function of $\mathbb{P}_i$, for a given direction set Δ, can be computed as follow:

$$\begin{aligned} \forall d \in \Delta,\ \delta_{\mathbb{P}_i}(d) &= \delta_{\mathbb{P}_{i-1} \sqcup F(\mathbb{P}_{i-1})}(d) \\ &= \max(\delta_{\mathbb{P}_{i-1}}(d), \delta_{A\mathbb{P}_{i-1} \oplus b}(d)) \\ &= \max(\delta_{\mathbb{P}_{i-1}}(d), \delta_{\mathbb{P}_{i-1}}(A^T d) + \langle b, d \rangle) \end{aligned}$$

This can be generalized to:

$$\delta_{\mathbb{P}_i}(d) = \max\left(\delta_{\mathbb{P}_{init}}(d), \delta_{\mathbb{P}_{init}}(A^{Tj}d) + \sum_{k=1}^{j}\langle b, A^{T(k-1)}d\rangle, j = 1, \ldots, i\right). \quad (4)$$

Note that in Equation 4, the support function of $\mathbb{P}_i$ are obtained using only support function of $\mathbb{P}_{init}$, which is the polyhedron obtained just before the execution of the loop, i.e. it is often the polyhedron representing the initialized variables. This allows us to compute efficiently $\delta_{\mathbb{P}_i}$ without having to deal with the complexity of the polyhedra abstract domain operations.

Remark that the technique presented in this section allows for a very efficient fixpoint computation for affine loops. Such loops are very common in embedded systems (for example linear filters are such loops) and are difficult to analyze. In particular, other acceleration techniques such as [15] are not able to handle affine loops, they can only compute the fixpoint for translations loop only. Our technique is much more general.

6 Experimentation

In this section, we apply Algorithm 1, presented so far, on some benchmark programs. We have implemented our framework on top of **Apron** [18] using the Parma Polyhedra Library (**PPL**) [3]. The experimentations are done on 2.4GHz Intel Core2 Duo laptop, with 8Gb of RAM.

Benchmarks Programs. Next, we present the results of our method on a collection of programs[1] implementing digital filters that are known to be hard to analyse using the standard polyhedra analysis. We used two filters inspired by [13] (named `filter1` and `filter2`) and five filters from the tests of the "Filter Verification Framework" software [11]. We choose filters of order 2 (`lp_iir_9600_2`) to 10 (`bs_iir_9600_12000_10_chebyshev`) to study how our method scales with the number of variables (a filter of order n has $2n + 2$ variables).

Comparison with Classical Kleene Iteration. Our benchmarks contain infinite loops without guards: in this case, it is hard to define thresholds for widening statically as techniques defined in [19] for example do not apply. So we analyse these programs using widening with delay (and no thresholds) on polyhedra abstract domain, with a delay of 15 iterations. The results are compared with ones obtained with our method. These results are given in Figure 4. In the "Program" column, $|V|$ denotes the number of variables of the program and $|\Delta|$ the number of chosen directions. In this table, "Yes" means that the analysis reaches a bounded fixpoint and $\top$ means an unbounded fixpoint. In this case, we give the execution time t. The sign $-$ means that the analysis did not terminate (with a time-out after 5 minutes). The results of Figure 4 shows that our method converges for all programs

[1] All programs can be found at `http://www.lix.polytechnique.fr/~{}bouissou/vmcai13`

Program			Widening		Our method	
Name	$\|V\|$	$\|\Delta\|$	Converging	t(s)	Converging	t(s)
`prog`	2	8	$\top$	0.091	Yes	0.029
`filter1`	6	24	$\top$	0.156	Yes	0.316
`filter2`	4	48	$\top$	0.053	Yes	0.672
`lp_iir_9600_2`	6	72	$\top$	0.208	Yes	0.049
`lp_iir_9600_4`	10	200	$\top$	6.563	Yes	0.167
`lp_iir_9600_4_elliptic`	10	200	–	–	Yes	0.308
`lp_iir_9600_6_elliptic`	14	392	–	–	Yes	2.564
`bs_iir_9600_12000_10_chebyshev`	22	968	–	–	Yes	19.780

Fig. 4. Results of analysis obtained using different methods

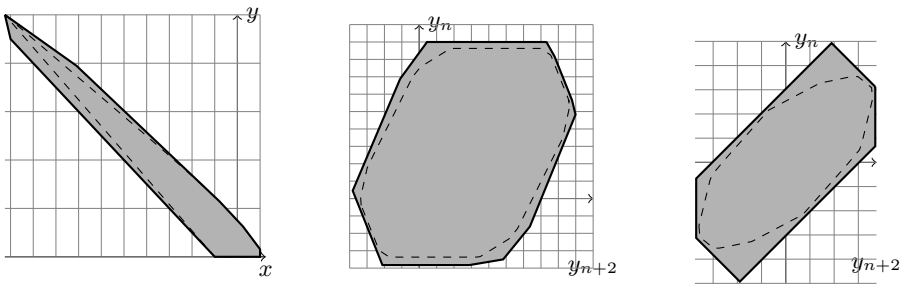
with a good execution time, where the widening fails to compute a bounded post-fixpoint. For these experiments, the direction sets we used are:

- `prog`, a randomly chosen direction set of 8 vectors.
- for `filter1` and `filter2`, we used the PCA analysis to determine a good directions set. We used 6 directions for `filter1` and 20 directions for `filter2`. Note that we also added to the directions set the box directions (i.e. $\pm X$ for each variable X) and for each direction d given by the PCA analysis, we added $-d$ to have a better coverage of the unit ball. The PCA analysis is done before programs analysis, and it is not taken into account in the execution time t.
- the octagonal directions for other programs.

Note that we also tried an analysis of these programs using the template domain, with a fixed template given by the directions set we used. In all cases, the analysis without widening did not terminate and widening with delay converged to $\top$.

In Figure 5, we show the post fixpoint we obtain for `prog` (with 100 directions), `filter2` and `lp_iir_9600_4`. We also show an under-approximation of the fixpoint, so the actual fixpoint is between both. This shows the quality of the invariant we compute.

Impact of the Acceleration Method. Finally, we want to stress out the importance of using an acceleration method to speed up the convergence of the algorithm. To do so, we compare the computation time and number of iterations of our algorithm with the same algorithm but with the identity as acceleration method, i.e. we stop when the sequence of support functions reaches its limit and not the accelerated sequence. As shown by the table of Figure 6, the use of a transformation method greatly improves the performance of the algorithm. We also compare two acceleration methods: the Δ^2-algorithm presented in Section 3.3 and the ε-algorithm. We see that both methods work well, the computation time being smaller for Δ^2 while the number of iterations needed to reach a post-fixpoint is smaller for the ε-algorithm. This was expected: the ε-algorithm is known to compute sequences that converge faster than Δ^2-algorithm, but its principle is that it repeatedly applies Δ^2-algorithm to the accelerated sequences. So its time and memory complexity are quadratic in the number of iteration,



(a) `prog`. (b) `filter2`. (c) `lp_iir_9600_4`.

Fig. 5. Results of the benchmarks programs. The filled polyhedron is the post fixpoint we obtain, the dashed on is the under-approximation of the fixpoint after, respectively, 100, 80 and 15 iterations of Kleene algorithm.

	Without acceleration		With Δ^2 method		With ε-method	
	t (s)	n_i	t (s)	n_i	t (s)	n_i
`lp_iir_9600_2`	0.058	47	0.037	16	0.047	17
`lp_iir_9600_4`	0.258	100	0.159	31	0.197	27
`lp_iir_9600_4_elliptic`	0.726	276	0.255	71	0.311	40
`lp_iir_9600_6_elliptic`	5.119	702	1.552	172	2.553	91
`bs_iir_9600_12000_10_chebyshev`	104.325	2391	19.873	524	-	-

Fig. 6. Influence of the acceleration method on the performance on execution time (t) and number of iterations (n_i)

while the complexity of Δ^2 is linear. This is the reason why the computation of the `bs_iir_9600_12000_10_chebyshev` program with the ε-algorithm timed out. Figure 6 shows that, even if theoretically neither the termination nor the acceleration is guaranteed, acceleration methods are in practice very useful for computing fixpoint of numerical programs.

7 Conclusion

Related work Improving the fixpoint computation in abstract interpretation has been a major concern since the beginning. Many techniques were proposed to improve Kleene iteration, like widening *with thresholds* [19], guided static analysis [16] or the use of acceleration techniques inspired from model checking [15]. Our method is complementary to these as it works well for numerical programs with a complex dynamics and for which no static constraints can be used to define relevant thresholds. As stated in the article, our technique finds its inspiration in the template abstract domain [24] and the use of support functions for hybrid systems analysis [20]. It however differs from an application of the abstract acceleration of [5] on the template domain as we follow the dynamics of the polyhedral analysis, which makes the result more precise than with the template domain. This was shown by our program `prog` in which our analysis

was able to prove that some part of the code is never reached while a template based analysis could not.

Conclusion and Future Work. In this article, we proposed a novel method to over-approximate the least fixpoint of the program semantics in the polyhedra domain. Our method uses a novel notion of convergence of convex sets, the abstract S-convergence, which is based on numerical and convex analysis tools. The algorithm we propose is very close to Kleene algorithm and its implementation does not require many changes in existing fixpoint solvers. One of the strengths of this method is that it can be tuned using many parameters (number and choice of the directions, ...) and thus offers a good trade-off between precision and computation time. Our experiments show that this method improves standard widening techniques on representative examples, as our prototype was able to compute a bounded post fixpoint for programs that the widening method was unable to bound.

Clearly, this method strongly depends on the choice of the directions to accelerate. We saw that the use of an apriori analysis using PCA and partial execution traces can help to determine relevant directions, the generalization and automatization of this process will be investigated. Also, our implementation is currently based on the ε-algorithm for accelerating the convergence of numerical sequences, we must consider other techniques in order to efficiently treat more kinds of programs. Our method is efficient to analyse affine programs, to generalize it to the non affine ones, we would like to experiment techniques of program linearisation. Finally, we would like to investigate the use of our technique to extrapolate on the dynamics of hybrid systems as defined in [20]: we believe that the abstract S-convergence can be used to approximate the limit of a dynamical system with unbounded horizon.

Acknowledgement. We want to thank A. Adjé and E. Goubault for their helpful suggestions and precious advices. The authors are also thankful to the anonymous reviewers for their helpful comments.

References

1. Amato, G., Parton, M., Scozzari, F.: Discovering invariants via simple component analysis. J. Symb. Comput. 47(12), 1533–1560 (2012)
2. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. Sci. Comput. Program. 58(1-2), 28–56 (2005), http://dx.doi.org/10.1016/j.scico.2005.02.003
3. Bagnara, R., Hill, P.M., Zaffanella, E.: Not necessarily closed convex polyhedra and the double description method. Formal Asp. Comput. 17(2), 222–257 (2005)
4. Bouissou, O., Seladji, Y.: Numerical abstract domain using support function. Presented at the Fifth International Workshop on Numerical Software Verification, http://www.lix.polytechnique.fr/ bouissou/pdf/bouissou_seladji_nsv_12.pdf
5. Bouissou, O., Seladji, Y., Chapoutot, A.: Acceleration of the abstract fixpoint computation in numerical program analysis. J. Symb. Comput. 47(12), 1479–1511 (2012)

6. Brezinski, C., Redivo Zaglia, M.: Extrapolation Methods-Theory and Practice. North-Holland (1991)
7. Brezinski, C., Redivo Zaglia, M.: Generalizations of Aitken's process for accelerating the convergence of sequences. Comp. and Applied Math. 26(2) (2007)
8. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, Invited Paper. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–97. ACM Press (1978)
10. 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 Press (1977)
11. Cox, A., Sankaranarayanan, S., Chang, B.-Y.E.: A Bit Too Precise? Bounded Verification of Quantized Digital Filters. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 33–47. Springer, Heidelberg (2012)
12. Delahaye, J.P., Germain-Bonne, B.: Résultats négatifs en accélération de la convergence. Numerische Mathematik 35, 443–457 (1980)
13. Feret, J.: Static Analysis of Digital Filters. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004)
14. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
15. Gonnord, L., Halbwachs, N.: Combining Widening and Acceleration in Linear Relation Analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
16. Gopan, D., Reps, T.W.: Guided Static Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer, Heidelberg (2007)
17. Hiriart-Urrut, J.B., Lemaréchal, C.: Fundamentals of Convex Analysis. Springer (2004)
18. 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)
19. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with Thresholds for Programs with Complex Control Graphs. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 492–502. Springer, Heidelberg (2011)
20. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Analysis: Hybrid Systems (2010)
21. Löhne, A., Zălinescu, C.: On convergence of closed convex sets. Journal of Mathematical Analysis and Applications 319(2), 617–634 (2006)
22. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
23. Rockafellar, R.: Convex analysis, vol. 28. Princeton Univ. Pr. (1997)
24. 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)
25. Sonntag, Y., Zălinescu, C.: Scalar convergence of convex sets. J. Math. Anal. Appl. 164(1), 219–241 (1992)
26. Wynn, P.: The epsilon algorithm and operational formulas of numerical analysis. Mathematics of Computation 15(74), 151–158 (1961)

SMT-Based Array Invariant Generation*

Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio

Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. This paper presents a constraint-based method for generating universally quantified loop invariants over array and scalar variables. Constraints are solved by means of an SMT solver, thus leveraging recent advances in SMT solving for the theory of non-linear arithmetic. The method has been implemented in a prototype program analyzer, and a wide sample of examples illustrating its power is shown.

Keywords: Program correctness, Invariant generation, SMT.

1 Introduction

Discovering loop invariants is an essential task for verifying the correctness of software. In particular, for programs manipulating arrays, usually one has to take into account invariant relationships among values stored in arrays and scalar variables. However, due to the unbounded nature of arrays, invariant generation for these programs is a challenging problem. In this paper we present a method for generating universally quantified loop invariants over array and scalar variables.

Namely, programs are assumed to consist of unnested loops and linear assignments, conditions and array accesses. Let $\overline{a} = (A_1, \ldots, A_m)$ be the array variables. Given an integer $k > 0$, our method generates invariants of the form:

$$\forall \alpha \ : \ 0 \leq \alpha \leq C(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0\,,$$

where $C, \mathcal{E}_{ij}, \mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables $\overline{v}$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$ for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, k\}$. This family of properties is quite general and allows handling a wide variety of programs.

Our method builds upon the so-called *constraint-based invariant generation* approach for discovering linear invariants [1], i.e., invariants expressed as linear inequalities over scalar variables. By means of Farkas' Lemma, the problem of the existence of an inductive invariant for a loop is transformed into a satisfiability problem in propositional logic over non-linear arithmetic. Despite the potential of the approach, its application has been limited so far due to the lack of good solvers for the obtained non-linear constraints.

However, recently significant progress has been made in SMT modulo the theory of non-linear arithmetic. In particular, the Barcelogic SMT solver has shown to be very effective on finding solutions in quantifier-free non-linear integer arithmetic [2]. These advances motivated us to revisit the constraint-based approach for linear invariants and extend it to programs with arrays.

* Partially supported by Spanish MEC/MICINN under grant TIN 2010-21062-C02-01.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 169–188, 2013.

2 Preliminaries

2.1 Transition Systems

Henceforth we will model programs by means of *transition systems*. A transition system $\mathcal{P} = \langle \overline{u}, \mathcal{L}, \ell_0, \mathcal{T} \rangle$ consists of a tuple of *variables* $\overline{u}$, a set of *locations* $\mathcal{L}$, an *initial location* ℓ_0 and a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is a triple $\langle \ell, \ell', \rho_\tau \rangle$, where $\ell, \ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and ρ_τ is the *transition relation*: a first-order Boolean formula over the program variables $\overline{u}$ and their primed versions $\overline{u}'$, which represent the values of the variables after the transition. In general, to every formula P (or expression E) over the program variables $\overline{u}$ we associate a formula P' (or expression E') which is the result of replacing every variable u_i in P (or E) by its corresponding primed version u_i'.

In this paper we will consider *scalar* variables, which take integer values, and *array* variables. We will denote scalar variables by $\overline{v}$ and array variables by $\overline{a}$. The *size* of an array $A \in \overline{a}$ is denoted by $|A|$ and the *domain* of its indices is $\{0 \dots |A| - 1\}$ (i.e., indices start at 0, as in C-like languages). We assume that arrays can only be indexed by expressions built over scalar variables. Hence, by means of the read/write semantics of arrays, we can describe transition relations as array equalities (possibly guarded by conjunctions of equalities and disequalities between scalar expressions) and quantified information of the form $\forall \alpha : 0 \leq \alpha \leq |A| - 1 \wedge P(\alpha) : A'[\alpha] = A[\alpha]$, where P does not depend on array variables. For example, Fig. 1 shows a program together with its transition system. A *path* π between two locations is associated to a transition

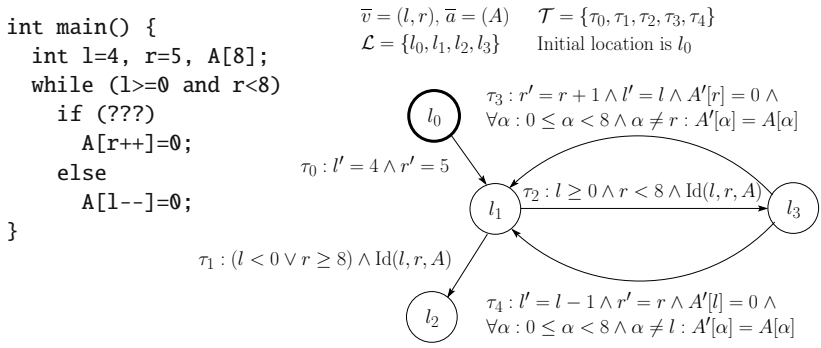


Fig. 1. Program and its transition system. Predicate $\mathrm{Id}(u_1, \dots, u_k)$ is short for $u_1 = u_1' \wedge \dots \wedge u_k = u_k'$, i.e., indicates those variables that remain identical after a transition.

relation ρ_π which is obtained by composition of the corresponding transitions relations. For instance, in the transition system in Fig. 1, the transition relations of the paths $\pi_0 = (l_0, \tau_0, l_1)$, $\pi_1 = (l_1, \tau_2, l_3, \tau_3, l_1)$ and $\pi_2 = (l_1, \tau_2, l_3, \tau_4, l_1)$ are:

$$\rho_{\pi_0} : l' = 4 \ \wedge \ r' = 5$$
$$\rho_{\pi_1} : l \geq 0 \ \wedge \ r < 8 \ \wedge \ r' = r + 1 \ \wedge \ l' = l \ \wedge \ A'[r] = 0 \ \wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq r \ : \ A'[\alpha] = A[\alpha]$$
$$\rho_{\pi_2} : l \geq 0 \ \wedge \ r < 8 \ \wedge \ l' = l - 1 \ \wedge \ r' = r \ \wedge \ A'[l] = 0 \ \wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \ \wedge \ \alpha \neq l \ : \ A'[\alpha] = A[\alpha] \, .$$

A path is *cyclic* if it contains a cycle. A set of locations $\mathcal{S}$ is a *cutset* if every cyclic path contains a location in $\mathcal{S}$. Locations in a cutset are *cutpoints*. In our example, paths π_1 and π_2 are cyclic, $\{l_1\}$ is a cutset and thus l_1 is a cutpoint.

Let $\mathcal{P}$ be a transition system with initial location ℓ_0, and $\mathcal{S}$ a cutset of $\mathcal{P}$. We call the *control-flow-graph of $\mathcal{P}$ induced by $\mathcal{S}$* the graph whose nodes are $\mathcal{N} = \{\ell_0\} \cup \mathcal{S}$, and such that for every path π_{ij} in the transition system connecting two locations ℓ_i and ℓ_j of $\mathcal{N}$ there exists a directed edge $\langle \ell_i, \ell_j, \pi_{ij} \rangle$. Note that therefore, every edge of the graph has an associated path in the transition system.

For a given strongly connected component (SCC) s of the control-flow-graph, its *initiation* paths are those paths in the transition system that label an edge from a location out of s to a location in s, and its *consecution* paths are those labeling an edge connecting only locations in s. For instance, the control-flow graph resulting from taking the cutset $\{l_1\}$ in our example has two nodes, l_0 and l_1, with one edge from l_0 to l_1 (π_0), and two self-edges at l_1 (π_1 and π_2). Thus, the SCC consisting of l_1 has one initiation path (π_0), and two consecution paths (π_1 and π_2).

2.2 Constraint-Based Invariant Generation

Here we review the *constraint-based invariant generation* approach [1]. Let us assume that we have selected all cutpoints, obtained all the SCCs and identified all respective initiation and consecution paths. The following well-known theorem establishes sufficient conditions for a set of properties to be invariant at the cutpoints:

Theorem 1. *Let $l_1^C, \ldots, l_p^C$ be a cutset of a SCC s. Let $P_1, \ldots, P_p$ be properties over the program variables $\overline{u}$ such that the following implications hold:*

i) for all initiation paths π^I from some l to some l_i^C: $\quad \forall \overline{u}, \overline{u}' \ \rho_{\pi^I} \quad \Rightarrow P_i'$
ii) for all consecution paths π^C from some l_j^C to some l_i^C: $\forall \overline{u}, \overline{u}' \ \rho_{\pi^C} \wedge P_j \Rightarrow P_i'$

Then $P_1, \ldots, P_p$ are invariant at $l_1^C, \ldots, l_p^C$. We say $P_1, \ldots, P_p$ are inductive invariants.

The idea of the constraint-based method is to consider a template for candidate invariant properties, e.g., linear inequalities in the scalar variables. These templates involve both program variables as well as parameters whose values are initially unknown and have to be determined so as to ensure invariance. To this end, the implications in Theorem 1 are expressed by means of *constraints* (hence the name of the approach) on the unknowns. If implications are encoded soundly, any solution to the constraints yields invariant properties for the cutpoints. In particular, if linear inequalities are taken as target invariants as in [1], implications can be transformed into arithmetic constraints over the unknowns by means of the following result from polyhedral geometry:

Theorem 2 (Farkas' Lemma [3]). *Consider a system S of linear inequalities $a_{i1}x_1 + \cdots + a_{in}x_n + b_i \leq 0$ $(i \in \{1, \ldots, m\})$ over real-valued variables $x_1, \ldots, x_n$. When S is satisfiable, it entails a linear inequality $c_1x_1 + \cdots + c_nx_n + d \leq 0$ iff there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_m$, such that $c_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \ldots, c_n = \sum_{i=1}^{m} \lambda_i a_{in}$, $d = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$. Further, S is unsatisfiable iff the inequality $1 \leq 0$ can thus be derived.*

Therefore, Farkas' Lemma allows one to transform an $\exists\forall$ problem into an $\exists$ problem. If all a_{ij} and b_i are known values, the resulting satisfiability problem is an SMT problem over linear arithmetic. Otherwise, an SMT problem over non-linear arithmetic is obtained. Moreover, if one is interested in linear invariants with integer coefficients, as some unknowns are integer (the invariant coefficients) and some are real (the multipliers $\lambda_0, \lambda_1, \ldots, \lambda_m$), an SMT problem in mixed arithmetic is obtained. However, as Farkas' Lemma applies to reals, one may lose some inductive invariants, namely those that only hold using the fact that the program variables are integers.

3 Array Invariants

In this section we present a constraint-based technique for generating array invariants for loop programs without nesting. Moreover, programs are assumed to contain linear expressions in assignments, `if` and `while` conditions, as well as in array accesses.

The idea of the method is, similarly as in [1], to express the conditions of Theorem 1 as algebraic constraints on the parameters of a prefixed invariant template. In order to provide the reader with intuition on how this is achieved, let us consider again the example in Fig. 1. In this program, an array A is filled with zeros from the middle outwards, moving alternatively to the left and to the right. Let us show that property $P \equiv \forall\alpha : 0 \leq \alpha < r - l - 1 : A[\alpha + l + 1] = 0$ is an inductive invariant for this program.

First of all, let us prove that initiation paths (namely, π_0) entail the property. In particular, we have to prove that $l' = 4 \wedge r' = 5 \rightarrow P'$.[1] This is trivial, since $l' = 4$ and $r' = 5$ imply that $r' - l' - 1$ is 0, i.e., the domain of the universally quantified variable α in P' is empty.

In general, our invariant generation method is aimed at universally quantified formulas, and we ensure that initiation paths imply the invariants by forcing that the domains of the universally quantified variables are empty.

Secondly, let us prove that consecution paths (i.e., π_1 and π_2) preserve the property. For example, for π_1 we have to prove that

$$P \wedge l \geq 0 \wedge r < 8 \wedge r' = r + 1 \wedge l' = l \wedge A'[r] = 0$$
$$\wedge \forall\alpha : 0 \leq \alpha < 8 \wedge \alpha \neq r : A'[\alpha] = A[\alpha] \rightarrow P' .$$

Now notice that the expression $r' - l' - 1$, which determines the domain of α in P', also has the property that $r' - l' - 1 = (r + 1) - l - 1 = (r - l - 1) + 1$. This means that, after π_1, the domain of α has exactly one new element, $\alpha = r - l - 1$. First, let us see that, after the path, property $A'[\alpha + l' + 1] = A'[\alpha + l + 1] = 0$ holds for the other values of α, i.e., $\alpha \in \{0, \ldots, r - l - 2\}$. Indeed this is the case: since $\forall\alpha : 0 \leq \alpha < 8 \wedge \alpha \neq r : A'[\alpha] = A[\alpha]$,

[1] From now on, program variables and their primed versions are universally quantified.

all positions of A' except for the r-th remain the same. But $A'[r] = A'[(r-l-1)+l'+1]$ precisely corresponds to $\alpha = r-l-1$. Hence from P we have that $A'[\alpha + l' + 1] = 0$ for all $\alpha \in \{0, \ldots, r-l-2\}$. Now we only need to prove $A'[\alpha + l' + 1] = 0$ for $\alpha = r-l-1$, which follows from the premise $A'[r] = 0$. In conclusion, P' holds.

In general, our invariant generation method will require that, after each consecution path, at most one new element is added to the domain of our universally quantified invariant, and that the contents of the arrays involved in the invariant are not changed after the path.

Back to the example, as regards π_2 we have to prove that

$$P \ \wedge \ l \geq 0 \ \wedge \ r < 8 \ \wedge \ l' = l - 1 \ \wedge \ r' = r \ \wedge \ A'[l] = 0$$
$$\wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq l \ : \ A'[\alpha] = A[\alpha] \ \rightarrow \ P' .$$

Again, the expression $r'-l'-1$ also satisfies that $r'-l'-1 = r-(l-1)-1 = (r-l-1)+1$. Hence the domain of α has exactly one new element. But unlike in the previous case, l changes its value. To prove P' from P, it is convenient to rewrite P so that array accesses are expressed in terms of $A[\alpha + l' + 1]$. By making a shift, P is equivalent to $\forall \alpha \ : \ 1 \leq \alpha < r'-l'-1 \ : \ A[\alpha+l'+1] = 0$. Again, since $\forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq l \ : \ A'[\alpha] = A[\alpha]$, all positions of A' except for the l-th remain the same. But $A'[l] = A'[l' + 1]$ precisely corresponds to $\alpha = 0$. Therefore $A'[\alpha+l'+1] = 0$ for all $\alpha \in \{1, \ldots, r'-l'-2\}$. Further, as $A'[l] = 0$, we have that $A'[\alpha + l' + 1] = 0$ for $\alpha = 0$. Thus P' holds.

Apart from proving that P is invariant, we may also want to check that the array accesses that occur in it are correct. As regards initiation paths, since the domain of α after π_0 is empty, there is nothing to check. Regarding consecution paths, for example for π_1 we have to see that

$$l \geq 0 \wedge r < 8 \wedge r' = r + 1 \wedge l' = l \rightarrow \forall \alpha : 0 \leq \alpha < r' - l' - 1 : \alpha + l' + 1 \geq 0 \wedge \alpha + l' + 1 < 8,$$

where for the sake of simplicity we have ignored the array variable. Now, given that array accesses are linear functions in α, it is sufficient to check correctness for $\alpha = 0$ and $\alpha = r'-l'-2$, i.e., that the above premises entail $l' + 1 \geq 0 \ \wedge \ l' + 1 < 8 \ \wedge \ r' - 1 \geq 0 \ \wedge \ r' - 1 < 8$. Let us assume that we have already looked for linear inequality invariants over scalar variables (e.g., with the techniques in [1,4]), and have found that $l \leq r - 1$ is a loop invariant. Adding this invariant to the transition relation suffices to prove the above implication. A similar argument applies for π_2.

In general, our invariant generation method guarantees that the array accesses occurring in the synthesized invariants are correct. As in the example, this is achieved by ensuring that the accesses of the extreme values of universally quantified variables are correct. Since this often requires arithmetic properties of the scalar variables of the program, in practice it is convenient that, prior to the application of our array invariant generation techniques, a linear relationship analysis for the scalar variables has already been carried out.

3.1 Invariant Generation for Programs with Arrays

Let $\overline{a} = (A_1, \ldots, A_m)$ be the tuple of array variables. Given a positive integer $k > 0$, our method generates invariants of the form

$$\forall \alpha \ : \ 0 \leq \alpha \leq C(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0 \, ,$$

where C, $\mathcal{E}_{ij}$ and $\mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables $\overline{v} = (v_1, \ldots, v_n)$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$, for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k\}$.

This template covers a quite general family of properties. See Sect. 5 for a sample of diverse programs for which we can successfully produce useful invariants and which cannot be handled by already existing techniques.

The invariant generation process at the cutpoint of the unnested loop under consideration is split into three steps, in order to make the approach computationally feasible:

1. Expressions C are generated such that the domain $\{0 \ldots C-1\}$ is empty after every initiation path reaching the cutpoint, and C does not change or is increased by one after every consecution path. This guarantees that any property universally quantified with this domain holds after all initiation paths and the domain includes at most one more element after every consecution path. We avoid the synthesis of different expressions that under the known information define the same domain. In the running example, we generate $C(l, r) = r - l - 1$.
2. For every expression C obtained in the previous step and for every array A_i, linear expressions $d_i\alpha + \mathcal{E}_i$ over the scalar variables are generated such that: (*i*) $A_i[d_i\alpha + \mathcal{E}_i]$ is a correct access for all α in $\{0 \ldots C-1\}$; (*ii*) none of the already considered positions in the quantified property is changed after any execution of the consecution paths; and (*iii*), after every consecution path, either $\mathcal{E}_i$ does not change or its value is $\mathcal{E}_i - d_i$. Namely, if C does not change, then $\mathcal{E}'_i = \mathcal{E}_i$ ensures that the invariant is preserved. Otherwise, the invariant has to be extended for a new value of α. If $\mathcal{E}_i$ does not change, from the previous condition for all $\alpha \in \{0, \ldots, C-1\}$ we have $A'_i[d_i\alpha + \mathcal{E}'_i] = A_i[d_i\alpha + \mathcal{E}_i]$. So we will try to extend the invariant with $\alpha = C$. Otherwise, if $\mathcal{E}'_i = \mathcal{E}_i - d_i$, then for all $\alpha \in \{1, \ldots, C\}$ we have $A'_i[d_i\alpha + \mathcal{E}'_i] = A_i[d_i(\alpha - 1) + \mathcal{E}_i]$. So we will try to extend the invariant with $\alpha = 0$.
   In the running example, we generate $d_{11} = 1$ and $\mathcal{E}_{11} = l + 1$.
3. For the selected C we choose k expressions $\mathcal{E}_{ij}$ for every array A_i among the generated $\mathcal{E}_i$, such that for each consecution path either all selected $\mathcal{E}_{ij}$ remain the same after the path, or all have as new value $\mathcal{E}_{ij} - d_{ij}$ after the path. Then, in order to generate invariant properties we just need to find integer coefficients a_{ij} and b_α and an expression $\mathcal{B}$ such that, depending on the case, either the property is fulfilled when $\alpha = C$ at the end of all consecution paths that increase C or it is fulfilled when $\alpha = 0$ at the end of all consecution paths that increase C. Further, $\mathcal{B}$ and b_α have to fulfill that the quantified property is maintained for $\alpha \in \{0 \ldots C-1\}$, assuming that the contents of the already accessed positions are not modified.
   For instance, in the running example for $k = 1$ we generate $a_{11} = 1$, $\mathcal{B} = b_\alpha = 0$, corresponding to the invariant $\forall \alpha : 0 \leq \alpha < r-l-1 : A[\alpha+l+1] \leq 0$; and $a_{11} = -1$, $\mathcal{B} = b_\alpha = 0$, corresponding to the invariant $\forall \alpha : 0 \leq \alpha < r-l-1 : -A[\alpha+l+1] \leq 0$.

Next we formalize all these conditions, which ensure that every solution to the last phase provides an invariant, and show how to encode them as SMT problems.

While for scalar linear templates the conditions of Theorem 1 can be directly transformed into constraints over the parameters [1], this is no longer the case for our template of array invariants. To this end we particularize Theorem 1 in a form that is suitable

for the constraint-based invariant generation method. The proof of this specialized theorem, given in detail below, mimics the proof of invariance of the running example given at the beginning of this section.

Let $\pi^I_1 \ldots \pi^I_p$ be the initiation paths to our cutpoint and $\pi^C_1 \ldots \pi^C_q$ the consecution paths going back to the cutpoint.

Theorem 3. *Let C, $\mathcal{B}$ and $\mathcal{E}_{ij}$ be linear polynomials with integer coefficients over the scalar variables, and a_{ij}, d_{ij}, $b_\alpha \in \mathbb{Z}$, for $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$. If*

1. *Every initiation path π^I_r with transition relation $\rho_{\pi^I_r}$ satisfies $\rho_{\pi^I_r} \Rightarrow C' = 0$.*
2. *For all consecution paths π^C_s with transition relation $\rho_{\pi^C_s}$, we have $\rho_{\pi^C_s} \Rightarrow (C' = C \vee C' = C + 1)$.*
3. *For all consecution paths π^C_s, all $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$, we have $\rho_{\pi^C_s} \wedge C' > 0 \Rightarrow 0 \le \mathcal{E}'_{ij} \le |A_i| - 1 \wedge 0 \le d_{ij}(C' - 1) + \mathcal{E}'_{ij} \le |A_i| - 1$.*
4. *For all consecution paths π^C_s we have either*
   (a) $\rho_{\pi^C_s} \wedge C' > 0 \Rightarrow \mathcal{E}'_{ij} = \mathcal{E}_{ij}$ for all $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$, or
   (b) $\rho_{\pi^C_s} \Rightarrow C' = C + 1 \wedge \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$ for all $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$.
5. *For all consecution paths π^C_s, we have $\rho_{\pi^C_s} \Rightarrow \forall \alpha : 0 \le \alpha \le C - 1 : A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] = A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ for all $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$.*
6. *For all consecution paths π^C_s, we have*
   - *$\rho_{\pi^C_s} \wedge C' = C + 1 \Rightarrow \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}C + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha C \le 0$, if case 4a applies.*
   - *$\rho_{\pi^C_s} \Rightarrow \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[\mathcal{E}'_{ij}] + \mathcal{B}' \le 0$, if case 4b applies.*
7. *For all consecution paths π^C_s, we have*
   - *$\rho_{\pi^C_s} \wedge 0 \le \alpha \le C - 1 \wedge x + \mathcal{B} + b_\alpha\alpha \le 0 \Rightarrow x + \mathcal{B}' + b_\alpha\alpha \le 0$ for some fresh universally quantified variable x, if case 4a applies.*
   - *$\rho_{\pi^C_s} \wedge 0 \le \alpha \le C - 1 \wedge x + \mathcal{B} + b_\alpha\alpha \le 0 \Rightarrow x + \mathcal{B}' + b_\alpha(\alpha + 1) \le 0$ for some fresh universally quantified variable x, if case 4b applies.*

Then $\forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha\alpha \le 0$ is invariant.

Proof. Following Theorem 1, we show that the property holds after each initiation path, and that it is maintained after each consecution path.

The first condition easily holds by applying 1, since we have that $\rho_{\pi^I_r} \Rightarrow C' = 0$ for every initiation path π^I_r, which implies $\forall \alpha : 0 \le \alpha \le C' - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$, since the domain of the quantifier is empty.

For the consecution conditions we have to show that for all consecution paths π^C_s, we have $\rho_{\pi^C_s} \wedge \forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha\alpha \le 0$ implies $\forall \alpha : 0 \le \alpha \le C' - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$.

By condition 2, we have $\rho_{\pi^C_s} \Rightarrow (C' = C \vee C' = C + 1)$, and by condition 4 either $\rho_{\pi^C_s} \wedge C' > 0 \Rightarrow \mathcal{E}'_{ij} = \mathcal{E}_{ij}$ for all $i \in \{1 \ldots m\}$, $j \in \{1 \ldots k\}$, or $\rho_{\pi^C_s} \Rightarrow C' = C + 1 \wedge \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$ for all $i \in \{1 \ldots m\}$, $j \in \{1 \ldots k\}$. We distinguish three cases:

1. $C' = C$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij}$. Then we have to ensure $\forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$. By condition 5, we can replace A'_i by A_i in the given domain, and hence we have to show that $\forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$. Then, since the array part coincides with

the one of the assumption, we can replace it in both places by some fresh variable x. Now it suffices to show that, assuming $x + \mathcal{B} + b_\alpha \alpha \leq 0$, we have $x + \mathcal{B}' + b_\alpha \alpha \leq 0$ for all value of x, which follows from the premises and condition 7.

2. $C' = C + 1$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. By conditions 1 and 2 we have $0 \leq C$, and hence $C = C' - 1$ belongs to the domain $\{0 \ldots C\}$ and $C' > 0$. Then, by condition 3, we have that $0 \leq d_{ij} C + \mathcal{E}_{ij} \leq |A_i| - 1 = |A'_i| - 1$ for all i and j. Therefore, we can extract the case $\alpha = C$ from the quantifier obtaining $\forall \alpha : 0 \leq \alpha \leq C - 1 : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ and $\Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij} C + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha C \leq 0$. The first part holds as before by the premises and conditions 5 and 7, and the second part holds by the premises and condition 6.
3. $C' = C + 1$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij} - d_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. Since, by conditions 1 and 2, we have $0 \leq C$, we have that C belongs to the domain $\{0 \ldots C\}$. By condition 3, we have $0 \leq \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij} \leq |A'_i| - 1$. Therefore, we can extract the case $\alpha = 0$ from the quantifier obtaining $\forall \alpha : 1 \leq \alpha \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij} - d_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha \alpha \leq 0$ and $\Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}_{ij} - d_{ij}] + \mathcal{B}' \leq 0$. For the first one, replacing α by $\alpha + 1$ we have $\forall \alpha : 1 \leq \alpha + 1 \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}(\alpha + 1) + \mathcal{E}_{ij} - d_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$, or equivalently $\forall \alpha : 0 \leq \alpha \leq C - 1 : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$, which holds by applying conditions 5 and 7 as before. The second part holds again by the premises and condition 6, using the fact that $\mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$. □

As we have described, our invariant generation method consists of three phases. The first phase looks for expressions C satisfying conditions 1 and 2. The second one provides, for every generated C and for every array A_i, expressions $\mathcal{E}_i$ with their corresponding integers d_i that fulfill conditions 3, 4 and 5. Note that, to satisfy condition 4, we need to record for each expression and path whether we have $\mathcal{E}'_i = \mathcal{E}_i$ or $\mathcal{E}'_i = \mathcal{E}_i - d_i$, so as to ensure that all expressions $\mathcal{E}_{ij}$ have the same behavior. Finally, in the third phase we have to find coefficients a_{ij} and b_α and an expression $\mathcal{B}$ fulfilling conditions 6 and 7.

Solutions to all three phases are obtained by encoding the conditions of Theorem 3 into SMT problems in non-linear arithmetic thanks to Farkas' Lemma. Note that, because of array updates, transition relations may not be conjunctions of literals (i.e., atomic predicates or negations of atomic predicates). As in practice the guarded array information is useless until the last phase, in the first two phases we use the unconditional part of a transition relation ρ, i.e., the part of ρ that is a conjunction of literals, denoted by $U(\rho)$.

3.2 Encoding Phase 1

Let C be $c_1 v_1 + \ldots + c_n v_n + c_{n+1}$, where $\overline{v}$ are the scalar variables and $\overline{c}$ are the integer unknowns. Then conditions 1 and 2 can be expressed as:

$$\exists \overline{c} \forall \overline{v}, \overline{v'} \bigwedge_{r=1}^{p} (U(\rho_{\pi_r^I}) \Rightarrow C' = 0) \wedge \bigwedge_{s=1}^{q} (U(\rho_{\pi_s^C}) \Rightarrow C' = C \vee C' = C + 1) .$$

We cannot apply Farkas' Lemma directly due to the disjunction in the conclusion of the second condition. To solve this, we move one of the two literals into the premise

and negate it. As the literal becomes a disequality, it can be split into a disjunction of inequalities. Finally, thanks to the distributive law, Farkas' Lemma can be applied and an existentially quantified SMT problem in non-linear arithmetic is obtained. We also encode the condition that each newly generated C must be different from all previously generated expressions at the cutpoint, considering all already known scalar invariants.

3.3 Encoding Phase 2

Here, for each C obtained in the previous phase and for each array A_i, we generate expressions $\mathcal{E}_i$ and integers d_i that satisfy conditions 3 and 5, and also condition 4 as a single expression and not combined with the other expressions.

The encoding of condition 3 is direct using Farkas' Lemma. Now let us sketch the encoding of condition 4. Let $\mathcal{E}_i$ be $e_1 v_1 + \ldots + e_n v_n + e_{n+1}$, where $\overline{e}$ are integer unknowns. Then, as $\mathcal{E}_i$ is considered in isolation, we need

$$\exists \overline{e}, d_i \; \forall \overline{v}, \overline{v'} \; \bigwedge_{s=1}^{q} \rho_{\pi_s^C} \Rightarrow ((C' = C + 1 \; \wedge \; \mathcal{E}'_i = \mathcal{E}_i - d_i) \; \vee \; C' \leq 0 \; \vee \; \mathcal{E}'_i = \mathcal{E}_i) \, .$$

To apply Farkas' Lemma, we use a similar transformation as for condition 2. In addition, it is imposed that the newly generated expressions are different from the previous ones.

Regarding condition 5, the encoding is rather different. In this case, for every consecution path π_s^C, array A_i and expression $G \Rightarrow A'_i[W] = M$ in $\rho_{\pi_s^C}$, we ensure that

$$\forall \alpha \; (\rho_{\pi_s^C} \; \wedge \; 0 \leq \alpha \leq C - 1 \; \wedge \; G \Rightarrow \; (W \neq d_i \alpha + \mathcal{E}_i \; \vee \; M = A_i[W])) \, .$$

To avoid generating useless expressions, we add in the encoding a condition stating that if $\mathcal{E}'_i = \mathcal{E}_i$ then for every consecution path where C is incremented, there is at least an access $A_i[W]$ in the path such that $W = d_i(C' - 1) + \mathcal{E}'_i$. Otherwise, i.e., if $\mathcal{E}'_i = \mathcal{E}_i - d_i$, then for every consecution path where C is incremented, there is at least an access $A_i[W]$ in the path such that $W = \mathcal{E}'_i$.

3.4 Encoding Phase 3

Condition 7 is straightforward. Regarding condition 6, the encoding does not need non-linear arithmetic, but requires to handle arrays:

$$\begin{array}{l} \exists \overline{a}, \overline{b}, b_\alpha \; \forall \overline{v}, \overline{v'}, \overline{A}, \overline{A'} \\ \bigwedge_{s=1}^{q} \; (\rho_{\pi_s^C} \qquad\qquad\quad \Rightarrow \; \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}'_{ij}] \; + \mathcal{B}' \; \leq \; 0) \qquad \wedge \\ \qquad (\rho_{\pi_s^C} \; \wedge \; C' = C + 1 \; \Rightarrow \; \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[C + \mathcal{E}'_{ij}] \; + \mathcal{B}' + b_\alpha C \; \leq \; 0) \, . \end{array}$$

Here, the use of the guarded array information is crucial. However, since we want to apply Farkas' Lemma, array accesses have to be replaced by new universally quantified integer variables. In order to avoid losing too much information, we add the array read semantics after the replacement; i.e., if $A[i]$ and $A[j]$ have been respectively replaced by fresh variables z_i and z_j, then $i = j \Rightarrow z_i = z_j$ is added.

4 Extensions

4.1 Relaxations on Domains

Let us consider the following program:

```
int A[2*N], min, max, i;
if (A[0] < A[1]) { min = A[0]; max = A[1]; }
else             { min = A[1]; max = A[0]; }
for (i = 2; i < 2*N; i += 2) {
  int tmpmin, tmpmax;
  if (A[i] < A[i+1]) { tmpmin = A[ i ]; tmpmax = A[i+1]; }
  else               { tmpmin = A[i+1]; tmpmax = A[ i ]; }
  if (max < tmpmax) max = tmpmax;
  if (min > tmpmin) min = tmpmin;  }
```

It computes the minimum and the maximum of an even-length array simultaneously, using a number of comparisons which is 1.5 times its length. To prove correctness, the invariants $\forall \alpha : 0 \leq \alpha \leq i - 1 : v[\alpha] \geq min$ and $\forall \alpha : 0 \leq \alpha \leq i - 1 : v[\alpha] \leq max$ are required. To discover them, two extensions of Theorem 3 are required:

- The domain of the universally quantified variable α cannot be forced to be initially empty. In this example, when the loop is entered, both invariants already hold for $\alpha = 0, 1$. This can be handled by applying our invariant generation method as described in Sect. 3.1, and for each computed invariant trying to extend the property for decreasing values of $\alpha = -1, -2$, etc. as much as possible. Finally, a shift of α is performed so that the domain of α begins at 0 and the invariant can be presented in the form of Sect. 3.1.
- The domain of the universally quantified variable α cannot be forced to increase at most by one at each loop iteration. For instance, in this example at each iteration the invariants hold for two new positions of the array. Thus, for a fixed parameter Δ, Condition 2 in Theorem 3 must be replaced by $\rho_{\pi_s^c} \Rightarrow (C' = C \ \vee \ C' = C + 1 \ \vee \ \cdots \ \vee \ C' = C + \Delta)$. In this example, taking $\Delta = 2$ is required. Further, conditions 4b, 6 and 7 must also be extended accordingly in the natural way.

4.2 Sorted Arrays

The program below implements binary search: given a non-decreasingly sorted array A and a value x, it determines whether there is a position in A containing x:

```
assume(N > 0);
int A[N], l = 0, u = N-1;
while (l <= u) {
  int m = (l+u)/2;
  if      (A[m] < x) l = m+1;
  else if (A[m] > x) u = m-1;
  else break;  }
```

To prove that, on exiting due to $l > u$, the property $\forall\alpha : 0 \le \alpha \le N-1 : A[i] \neq x$ holds, one can use that $\forall\alpha : 0 \le \alpha \le l-1 : A[\alpha] < x$ and $\forall\alpha : u+1 \le \alpha \le N-1 : A[\alpha] > x$ are invariant. To synthesize them, the fact that A is sorted must be taken into account. The following theorem results from incorporating the property of sortedness into Theorem 3:

Theorem 4. *Let C, $\mathcal{B}$ and $\mathcal{E}_{ij}$ be linear polynomials with integer coefficients over the scalar variables, and a_{ij}, d_{ij}, $b_\alpha \in \mathbb{Z}$, for $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$. If*

1. *For all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$ we have $b_\alpha \ge 0$, and $d_{ij} > 0 \Rightarrow a_{ij} \ge 0$, and $d_{ij} < 0 \Rightarrow a_{ij} \le 0$.*
2. *Each initiation path π_r^I with transition relation $\rho_{\pi_r^I}$ fulfills $\rho_{\pi_r^I} \Rightarrow C' = 0$.*
3. *Each initiation path π_r^I with transition relation $\rho_{\pi_r^I}$ fulfills $\rho_{\pi_r^I} \Rightarrow \forall\beta : 0 < \beta \le |A_i'| - 1 : A_i'[\beta - 1] \le A_i'[\beta]$ for all $i \in \{1 \dots m\}$.*
4. *Each consecution path π_s^C with transition relation $\rho_{\pi_s^C}$ fulfills $\rho_{\pi_s^C} \Rightarrow C' \ge C$.*
5. *For all consecution paths π_s^C all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$ we have $\rho_{\pi_s^C} \wedge C' > 0 \Rightarrow 0 \le \mathcal{E}'_{ij} \le |A_i| - 1 \wedge 0 \le d_{ij}(C'-1) + \mathcal{E}'_{ij} \le |A_i| - 1$.*
6. *For all consecution paths π_s^C we have one of the following:*
   (a) *$\rho_{\pi_s^C} \wedge C' > 0 \wedge a_{ij} > 0 \Rightarrow \mathcal{E}'_{ij} \le \mathcal{E}_{ij}$ and $\rho_{\pi_s^C} \wedge C' > 0 \wedge a_{ij} < 0 \Rightarrow \mathcal{E}'_{ij} \ge \mathcal{E}_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$;*
   (b) *$\rho_{\pi_s^C} \Rightarrow C' > C$ and $\rho_{\pi_s^C} \wedge a_{ij} > 0 \Rightarrow \mathcal{E}'_{ij} \le \mathcal{E}_{ij} - (C'-C)d_{ij}$ and $\rho_{\pi_s^C} \wedge a_{ij} < 0 \Rightarrow \mathcal{E}'_{ij} \ge \mathcal{E}_{ij} - (C'-C)d_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$.*
7. *For all consecution paths π_s^C, we have $\rho_{\pi_s^C} \Rightarrow \forall\beta : 0 \le \beta \le |A_i| - 1 : A_i'[\beta] = A_i[\beta]$ for all $i \in \{1 \dots m\}$.*
8. *For all consecution paths π_s^C, we have*
   - *$\rho_{\pi_s^C} \wedge C' > C \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}(C'-1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(C'-1) \le 0$, if case 6a applies.*
   - *$\rho_{\pi_s^C} \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}(C'-C-1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(C'-C-1) \le 0$, if case 6b applies.*
9. *For all consecution paths π_s^C, we have*
   - *$\rho_{\pi_s^C} \wedge 0 \le \alpha \le C-1 \wedge x + \mathcal{B} + b_\alpha\alpha \le 0 \Rightarrow x + \mathcal{B}' + b_\alpha\alpha \le 0$ for some fresh universally quantified variable x, if case 6a applies.*
   - *$\rho_{\pi_s^C} \wedge 0 \le \alpha \le C-1 \wedge x + \mathcal{B} + b_\alpha\alpha \le 0 \Rightarrow x + \mathcal{B}' + b_\alpha(\alpha + C' - C) \le 0$ for some fresh universally quantified variable x, if case 6b applies.*

Then $\forall\alpha : 0 \le \alpha \le C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha\alpha \le 0$ is invariant.

Proof. First of all, let us remark that arrays are always sorted in non-decreasing order, and that their contents are never changed. This follows by induction from conditions 3 and 7. Moreover, it can also be seen from conditions 2 and 4 that $C \ge 0$ is an invariant property.

Now, we will show that the property in the statement of the theorem holds after every initiation path reaching our cutpoint and that it is maintained after every consecution path going back to the cutpoint.

The first condition easily holds applying 2, since we have that $\rho_{\pi_r^I} \Rightarrow C' = 0$ for every initiation path π_r^I, which implies $\forall\alpha : 0 \le \alpha \le C'-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$, since the domain of the quantifier is empty.

For the consecution conditions we have to show that for all consecution paths π_s^C, we have $\rho_{\pi_s^C} \wedge \forall \alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ implies $\forall \alpha : 0 \leq \alpha \leq C'-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$.

By condition 4, we have that $\rho_{\pi_s^C} \Rightarrow C' \geq C$. We distinguish three cases:

1. $C' = C$ and case 6a holds. If $C' = 0$ there is nothing to prove. Otherwise $C' > 0$, and by hypothesis we have that $\forall \alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$. Together with $\rho_{\pi_s^C}$, this implies $\forall \alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ by instantiating appropriately x in condition 9.
   Now, let us show that for all $i \in \{1 \ldots m\}$, for all $j \in \{1 \ldots k\}$ and for all $\alpha \in \{0 \ldots C-1\}$ we have $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$. Let us consider three subcases:
   - $a_{ij} > 0$. Then $\mathcal{E}'_{ij} \leq \mathcal{E}_{ij}$ by condition 6. Hence for all $\alpha \in \{0 \ldots C-1\}$ we have $d_{ij}\alpha + \mathcal{E}'_{ij} \leq d_{ij}\alpha + \mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$.
   - $a_{ij} < 0$. Then $\mathcal{E}'_{ij} \geq \mathcal{E}_{ij}$ by condition 6. Hence for all $\alpha \in \{0 \ldots C-1\}$ we have $d_{ij}\alpha + \mathcal{E}'_{ij} \geq d_{ij}\alpha + \mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \geq A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order (note that, by condition 5, we have that $0 \leq d_{ij}\alpha + \mathcal{E}'_{ij} \leq |A_i| - 1 = |A'_i| - 1$, so array accesses are within bounds). Therefore $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$.
   - $a_{ij} = 0$. Then the inequality trivially holds.

   Altogether we have that $\forall \alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. Now our goal easily follows, taking into account that $C' = C$ and that by condition 7 we can replace A_i by A'_i.
2. $C' > C$ and case 6a holds. Then $C' > 0$, and following the same argument as in the previous case we get that $\forall \alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$, where A_i has been replaced by A'_i by virtue of condition 7.
   It only remains to prove that $\forall \alpha : C \leq \alpha \leq C'-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ (note that, by condition 5, we have that $0 \leq \mathcal{E}'_{ij} \leq |A'_i| - 1$ and $0 \leq d_{ij}(C'-1) + \mathcal{E}'_{ij} \leq |A'_i| - 1$, so again array accesses are within bounds). To this end, let us consider $\alpha \in \{C \ldots C'-1\}$ and let us show that $a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}]$ for all $i \in \{1 \ldots m\}$ and for all $j \in \{1 \ldots k\}$. We distinguish three cases:
   - $d_{ij} > 0$. Then $\alpha \leq C'-1$ implies $d_{ij}\alpha \leq d_{ij}(C'-1)$, and hence $d_{ij}\alpha + \mathcal{E}'_{ij} \leq d_{ij}(C'-1) + \mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \geq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} < 0$. Then $\alpha \leq C'-1$ implies $d_{ij}\alpha \geq d_{ij}(C'-1)$, and hence $d_{ij}\alpha + \mathcal{E}'_{ij} \geq d_{ij}(C'-1) + \mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \geq A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \leq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} = 0$. The goal trivially holds.

   Thus $\Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' \leq \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}] + \mathcal{B}'$. Now, by condition 1 we have $b_\alpha \geq 0$, hence $\alpha \leq C'-1$ implies $b_\alpha \alpha \leq b_\alpha (C'-1)$. Therefore $\Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha (C'-1) \leq 0$ by condition 8.

3. $C' > C$ and case 6b holds (notice that $C' = C$ and case 6b together are not possible). By hypothesis we have $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}\alpha+\mathcal{E}_{ij}]+\mathcal{B}+b_\alpha\alpha \leq 0$. Together with $\rho_{\pi_s^C}$, this implies $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}\alpha+\mathcal{E}_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ by instantiating appropriately x in condition 9. By shifting the universally quantified variable the previous formula can be rewritten as $\forall\alpha : C'-C \leq \alpha \leq C'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]+\mathcal{B}'+b_\alpha(\alpha-(C'-C)) \leq 0$.
Now, let us show that for all $i \in \{1\ldots m\}$, for all $j \in \{1\ldots k\}$ and for all $\alpha \in \{C'-C\ldots C'-1\}$ we have $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$. Let us consider three subcases:
   - $a_{ij} > 0$. Then $\mathcal{E}'_{ij} \leq \mathcal{E}_{ij}-(C'-C)d_{ij}$ by condition 6. Hence for all $\alpha \in \{C'-C\ldots C'-1\}$ we have $d_{ij}\alpha+\mathcal{E}'_{ij} \leq d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$.
   - $a_{ij} < 0$. Then $\mathcal{E}'_{ij} \geq \mathcal{E}_{ij}-(C'-C)d_{ij}$ by condition 6. Hence for all $\alpha \in \{C'-C\ldots C'-1\}$ we have $d_{ij}\alpha+\mathcal{E}'_{ij} \geq d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \geq A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$.
   - $a_{ij} = 0$. Then the inequality trivially holds.

   Altogether we have that $\forall\alpha : C'-C \leq \alpha \leq C'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$, where A_i has been replaced by A'_i by virtue of condition 7.
   It only remains to prove that $\forall\alpha : 0 \leq \alpha \leq C'-C-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ (note that, by condition 5, we have that $0 \leq \mathcal{E}'_{ij} \leq |A'_i|-1$ and $0 \leq d_{ij}(C'-1)+\mathcal{E}'_{ij} \leq |A'_i|-1$, so again array accesses are within bounds). To this end, let us consider $\alpha \in \{0\ldots C'-C-1\}$ and let us show that $a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]$ for all $i \in \{1\ldots m\}$ and for all $j \in \{1\ldots k\}$. We distinguish three cases:
   - $d_{ij} > 0$. Then $\alpha \leq C'-C-1$ implies $d_{ij}\alpha \leq d_{ij}(C'-C-1)$, and hence $d_{ij}\alpha+\mathcal{E}'_{ij} \leq d_{ij}(C'-C-1)+\mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \geq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} < 0$. Then $\alpha \leq C'-C-1$ implies $d_{ij}\alpha \geq d_{ij}(C'-C-1)$, and hence $d_{ij}\alpha+\mathcal{E}'_{ij} \geq d_{ij}(C'-C-1)+\mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \geq A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \leq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} = 0$. The goal trivially holds.

   Thus $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}' \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]+\mathcal{B}'$. Now, by condition 1 we have $b_\alpha \geq 0$, hence $\alpha \leq C'-C-1$ implies $b_\alpha\alpha \geq b_\alpha(C'-C-1)$. Therefore $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}(C'-C-1)\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha(C'-C-1) \leq 0$ by condition 8.

□

By means of the previous theorem, (an equivalent version of) the desired invariants can be discovered. However, to the best of our knowledge, results on the synthesis of invariants for programs with sorted arrays are not reported in the literature. See Sect. 5 for other examples that can be handled by means of this extension.

5 Experimental Evaluation

The method presented in Sects. 3 and 4 has been implemented in the tool CppInv[2]. For solving the generated constraints, we use the Barcelogic SMT solver [5]. As discussed in Sect. 2.2, after applying Farkas' Lemma an SMT problem for mixed non-linear arithmetic is obtained. For this theory, Barcelogic has proved to be very effective in finding solutions [2]; e.g., it won the division of quantifier-free non-linear integer arithmetic (QF_NIA) in the 2009 edition of the SMT-COMP competition (www.smtcomp.org/2009), and since then no other competing solver in this division has solved as many problems.

In addition to the examples already shown in this paper, CppInv automatically generates array invariants for a number of different programs. The following table shows some of them, together with the corresponding loop invariants:

Heap property:

```
const int N;
assume(N >= 0);
int A[2*N], i;
for (i = 0; 2*i+2 < 2*N; ++i)
  if (A[i]>A[2*i+1] or A[i]>A[2*i+2])
    break;
```

Loop invariants:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] \leq A[2\alpha+2]$
$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] \leq A[2\alpha+1]$

Partial initialization [6]:

```
const int N;
assume(N >= 0);
int A[N], B[N], C[N], i, j;
for (i = 0, j = 0; i < N; ++i)
  if (A[i] == B[i])
    C[j++] = i;
```

Loop invariant:

$\forall \alpha : 0 \leq \alpha \leq j-1 : C[\alpha] \leq \alpha + i - j$
$\forall \alpha : 0 \leq \alpha \leq j-1 : C[\alpha] \geq \alpha$

Array palindrome:

```
const int N;
assume(N >= 0);
int A[N], i;
for (i = 0; i < N/2; ++i)
  if (A[i] != A[N-i-1]) break;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = A[N-\alpha-1]$

Array initialization [6]:

```
const int N;
assume(N >= 0);
int A[N], i;
for (i = 0; i < N; ++i)
  A[i] = 2*i+3;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 2\alpha + 3$

Array insertion:

```
const int N;
int A[N], i, x, j;
assume(0 <= i and i < N);
for (x = A[i], j = i-1;
     j >=0 and A[j] > x; --j)
  A[j+1] = A[j];
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-j-2 : A[i-\alpha] \geq x+1$

Sequential initialization [7]:

```
const int N;
assume(N > 0);
int A[N], i;
for (i = 1, A[0] = 7; i < N; ++i)
  A[i] = A[i-1] + 1;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-2 : \ A[\alpha+1] = A[\alpha] + 1$

[2] The tool, together with a sample of example programs it can analyze, can be downloaded at www.lsi.upc.edu/~albert/cppinv-bin.tar.gz.

Array copy [7]:

```
const int N;
assume(N >= 0);
int A[N], B[N], i;
for (i = 0; i < N; ++i)
  A[i] = B[i];
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = B[\alpha]$

First not null [7]:

```
const int N;
assume(N >= 0);
int A[N], s, i;
for (i = 0, s = N; i < N; ++i)
  if (s == N and A[i] != 0) {
    s=i;
    break;
  }
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 0$

Array partition [8]:

```
const int N;
assume(N >= 0);
int A[N], B[N], C[N], a, b, c;
for (a=0, b=0, c=0; a < N; ++a)
  if (A[a] >= 0) B[b++]=A[a];
  else           C[c++]=A[a];
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq b-1 : B[\alpha] \geq 0$
$\forall \alpha : 0 \leq \alpha \leq c-1 : C[\alpha] < 0$

Array maximum [7]:

```
const int N;
assume(N > 0);
int A[N], i, max;
for (i = 1, max = A[0]; i < N; ++i)
  if (max < A[i])
    max = A[i];
```

Loop invariant:

$\forall \alpha : 0 \leq \alpha \leq i-1 : A[\alpha] \leq max$

First occurrence:

```
const int N;
assume(N > 0);
int A[N], x = getX(), l, u;
// A is sorted in ascending order
for (l = 0, u = N; l < u; ) {
  int m = (l+u)/2;
  if (A[m] < x) l = m+1;
  else u = m;   }
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq l-1 : A[\alpha] < x$
$\forall \alpha : 0 \leq \alpha \leq N-1-u : A[N-1-\alpha] \geq x$

Sum of pairs:

```
const int N;
assume(N > 0);
int A[N], x = getX(), l = 0, u = N-1;
// A is sorted in ascending order
while (l < u)
  if      (A[l] + A[u] < x) l = l+1;
  else if (A[l] + A[u] > x) u = u-1;
  else break;
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq l-1 : A[\alpha] + A[u] < x$
$\forall \alpha : 0 \leq \alpha \leq N-u-2 : A[N-1-\alpha] + A[l] > x$

As a final experiment, we have run CppInv over a collection of programs written by students. It consists of 38 solutions to the problem of finding the first occurrence of an element in a sorted array of size N in $O(\log N)$ time. These solutions have been taken from the online learning environment for computer programming Jutge.org (see www.jutge.org), which is currently being used in several programming courses in the Universitat Politècnica de Catalunya. The benchmark suite corresponds to all submitted iterative programs that have been accepted, i.e., such that for all input tests the output matches the expected one. These programs can be considered more realistic code than the examples above (**First occurrence** program), since most often they are not the most elegant solution but a working one with many more conditional statements than necessary. For example, here is an instance of such a program:

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);
  int e = 0, d = N - 1, m, pos;
  bool found = false, exit  = false;
  while (e <= d and not exit) {
    m = (e+d)/2;
    if      (x > A[m]) {
      if (not found) e = m+1;
      else exit = true;
    }
    else if (x < A[m]) {
      if (not found) d = m-1;
      else exit = true;
    }
    else {
      found = true; pos = m; d = m-1;
    } }
  if (found) {
    while (x  == A[pos-1]) --pos;
    return pos; }
  return -1; }
```

This particular example is interesting because, with the aid of our tool, we realized that it does not work in $O(\log N)$ time as required, and is thus a false positive. Namely, our tool produces the following invariants for the first loop:

$$\forall \alpha : \quad 0 \leq \alpha \leq e - 1 \ : A[\alpha] < x \,,$$
$$\forall \alpha : d + 1 \leq \alpha \leq N - 1 : A[\alpha] \geq x \,.$$

By manual inspection one can see that $found \rightarrow (A[pos] = x \wedge d = pos - 1)$ and $exit \rightarrow found$ are also invariant. Therefore, if on exit of the loop the property $e \leq d$ holds, then *exit* and *found* are true and, with all this information, it is unknown whether the contents of the array between e and $pos - 1$ are equal to x. Since this segment can be arbitrarily long, the second loop may take $O(N)$ time to find the first occurrence of x. This reasoning allowed us to cook an input for which indeed the program behaves linearly. On the other hand, by means of the generated invariants it can be seen that the problem is that the loop may be exited too early, and that by replacing in the first loop the body of the first conditional by `e = m+1` and the second one by `d = m-1`, the program becomes correct and meets the complexity requirements.

In general, for all programs in the benchmark suite our tool was able to find automatically both standard invariants. The time consumed was very different depending on how involved the code was. Anyway, the main problem as regards efficiency is that in its current form our prototype exhaustively generates first all scalar invariants and then, using all of them, generates all array invariants. Further work is needed to heuristically guide the search of scalar invariants, so that only useful information is inferred.

We also applied our tool to some of the submissions rejected by Jutge.org. In some cases the generated invariants helped us to fix the program. E.g., for the following code:

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);
  int i = 0, j = N-1;
  while (i <= j) {
    if (x == A[i]) return  i;
    if (x < A[i])  return -1;
    int m = (i+j)/2;
    if (x < A[m]) j = m-1;
    else          i = m+1; }
  return -1; }
```

In this case, the generated invariants are:

$$\begin{aligned} \forall \alpha : \quad & 0 \le \alpha \le i-1 \quad : A[\alpha] \le x\,, \\ \forall \alpha : \; & j+1 \le \alpha \le N-1 : A[\alpha] > x\,. \end{aligned}$$

One may notice that the first invariant should have a strict inequality, and that this problem may be due to a wrong condition in the last conditional. Indeed, by replacing the condition $x < A[m]$ by $x \le A[m]$, we obtain a set of invariants that allow proving the correctness of the program.

6 Related Work

There is a remarkable amount of work in the literature aimed at the synthesis of quantified invariants for programs with arrays. Some of the techniques fall into the framework of *abstract interpretation* [9]. In [6], the index domain of arrays is partitioned into several symbolic intervals I, and then each subarray $A[I]$ is associated to a summary auxiliary variable A_I. Although assignments to individual array elements can thus be handled precisely, in order to discover relations among the contents at different indices, hints must be manually provided. This shortcoming is overcome in [7], where additionally relational abstract properties of summary variables and shift variables are introduced to discover invariants of the form $\forall \alpha : \alpha \in I : \psi(A_1[\alpha + k_1], ..., A_m[\alpha + k_m], \overline{v})$, where $k_1, \ldots, k_m \in \mathbb{Z}$ and $\overline{v}$ are scalar variables. In comparison with our techniques, the previous approaches force all array accesses to be of the form $\alpha + k$. As a consequence, programs like **Array palindrome** or **Heap property** (see Sect. 5) cannot be handled. Moreover, the universally quantified variable is not allowed to appear outside array accesses. For this reason, our analysis can be more precise, e.g., in the **Array initialization** and the **Partial initialization** [6] examples. Another technique based on abstract interpretation is presented in [10]. While their approach can discover more general properties than ours, it requires that the user provides templates to guide the analysis. Yet another abstract interpretation-based method is given in [11], where *fluid updates* are defined on a symbolic heap abstraction.

Predicate abstraction techniques [12] can also be seen as instances of abstract interpretation. Here, a set of predefined predicates is considered, typically provided manually by the user or computed heuristically from the program code and the assertions to be proved. Then one generates an invariant built only over those predicates. This track of research was initiated in [13], where by introducing Skolem constants, universally-quantified loop invariants for arrays can be discovered. In [14], it is shown how the strongest universally quantified inductive invariant over the given predicates can be generated. Further works integrate predicate abstraction into the CEGAR loop [15,16], apply algorithmic learning [17] or discover invariants with complex pre-fixed Boolean structure [18]. Unlike most of these predicate abstraction-based techniques, our approach does not require programs to be annotated with assertions, thus allowing one to analyze code embedded into large programs, or with predicates, which sometimes require ingenuity from the user. To alleviate the need of supplying predicates, in [19] *parametric predicate abstraction* was introduced. However, the properties considered there express relations between all elements of two data collections, while our approach is able to express pointwise relations.

Another group of techniques is based on *first-order theorem proving*. In [20,21], the authors generate invariants with alternations of quantifiers for loop programs without nesting. First, one describes the loop dynamics by means of first-order formulas, possibly using additional symbols denoting array updates or loop counters. Then a saturation theorem prover eliminates auxiliary symbols and reports the consequences without these symbols, which are the invariants. One of the problems of the method is the limited capability of arithmetic reasoning of the theorem prover (as opposed to SMT solvers, where arithmetic reasoning is hard-wired in the theory solvers). In [22] a related approach is presented, where invariants are generated by examining candidates supplied by an interpolating theorem prover. In addition to suffering from similar arithmetic reasoning problems as [20], the approach also requires program assertions.

Other methods use *computational algebra*, e.g., [23]. One of the limitations of [23] is that array variables are required to be either write-only or read-only. Hence, unlike our method, programs such as **Sequential initialization** [7] and **Array insertion** (see Sect. 5) cannot be handled.

Finally, the technique that has been presented in this paper belongs to the *constraint-based* methods. In this sense it is related to that in [24]. There, the authors present a constraint-based algorithm for the synthesis of invariants expressed in the combined theory of linear integer arithmetic (LI) and uninterpreted function symbols (UIF). By means of the reduction of the array property fragment to LI+UIF, it is claimed that the techniques can be extended for the generation of universally quantified invariants for arrays. However, the language of our invariants is outside the array property fragment, since we can generate properties where indices do not necessarily occur in array accesses (e.g., see the **Array initialization** or the **Partial initialization** examples in Sect. 5). Finally, the technique in [24] is applied in [8] to generating path invariants in the context of the CEGAR loop. As the framework in [8] is independent of any concrete invariant generation technique, we believe that our method could be used as an alternative in a portfolio approach to path invariant-based program analysis.

7 Conclusions and Future Work

In short, the contributions of this paper are:

- *a new constraint-based method for the generation of universally quantified invariants of array programs.* Unlike other techniques, it does not require extra predicates nor assertions. It does not need the user to provide a template either, but it can take advantage of hints by partially instantiating the global template considered here.
- *extensions of the approach for sorted arrays.* To our knowledge, results on the synthesis of invariants for programs with sorted arrays are not reported in the literature.
- *an implementation of the presented techniques that is freely available.* The constraint solving engine of our prototype depends on SMT. Hence, our techniques will directly benefit from any further advances in SMT solving.

For future work, we plan to extend our approach to a broader class of programs. As a first step we plan to relax Theorem 3, so that, e.g., overwriting on positions in which the

invariant already holds is allowed. We would also like to handle nested loops, so that for instance sorting algorithms can be analyzed. Another line of work is the extension of the family of properties that our approach can discover as invariants. E.g., a possibility could be considering disjunctive properties, or allowing quantifier alternation. The former allows analyzing algorithms such as sentinel search, while the latter is necessary to express that the output of a sorting algorithm is a permutation of the input.

Moreover, the invariants that our method generates depend on the coefficients and expressions obtained in each of its three phases, which in turn depend on the previous linear relationship analysis of scalar variables. We leave for future research to study how to make the approach resilient to changes in the outcome of the different phases.

Finally, as pointed out in Sect. 5, the efficiency of CppInv can be improved. In particular, further work is needed to heuristically guide the search of scalar invariants, so that only useful information is inferred.

Acknowledgments. We would like to thank Jutge.org team for providing us with programs written by students. We are also grateful to the anonymous referees of a previous version of this paper for their helpful comments.

References

1. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear Invariant Generation Using Non-linear Constraint Solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
2. Borralleras, C., Lucas, S., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: SAT Modulo Linear Arithmetic for Solving Polynomial Constraints. J. Autom. Reasoning 48(1), 107–131 (2012)
3. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (June 1998)
4. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL, pp. 84–96 (1978)
5. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The Barcelogic SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008)
6. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 338–350. ACM (2005)
7. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 339–348. ACM (2008)
8. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 300–309. ACM (2007)
9. 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)
10. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 235–246. ACM (2008)
11. Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
12. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
13. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)

14. Lahiri, S.K., Bryant, R.E.: Constructing Quantified Invariants via Predicate Abstraction. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 267–281. Springer, Heidelberg (2004)
15. 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)
16. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012)
17. Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 328–343. Springer, Heidelberg (2010)
18. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Hind, M., Diwan, A. (eds.) PLDI, pp. 223–234. ACM (2009)
19. Cousot, P.: Verification by Abstract Interpretation. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 243–268. Springer, Heidelberg (2004)
20. Kovács, L., Voronkov, A.: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
21. Hoder, K., Kovács, L., Voronkov, A.: Case Studies on Invariant Generation Using a Saturation Theorem Prover. In: Batyrshin, I., Sidorov, G. (eds.) MICAI 2011, Part I. LNCS, vol. 7094, pp. 1–15. Springer, Heidelberg (2011)
22. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
23. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for Arrays (Tool Paper). In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 348–356. Springer, Heidelberg (2010)
24. 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)

Flow-Sensitive Fault Localization

Jürgen Christ[1], Evren Ermis[1], Martin Schäf[2,⋆], and Thomas Wies[3]

[1] University of Freiburg
[2] United Nations University, IIST, Macau
[3] New York University

Abstract. Identifying the cause of an error is often the most time-consuming part in program debugging. Fault localization techniques can help to automate this task. Particularly promising are static proof-based techniques that rely on an encoding of error traces into *trace formulas*. By identifying irrelevant portions of the trace formula, the possible causes of the error can be isolated. One limitation of these approaches is that they do not take into account the control flow of the program and therefore miss common causes of errors, such as faulty branching conditions. This limitation is inherent to the way the error traces are encoded. In this paper, we present a new flow-sensitive encoding of error traces into trace formulas. The new encoding enables proof-based techniques to identify irrelevant conditional choices in an error trace and to include a justification for the truth value of branching conditions that are relevant for the localized cause of an error. We apply our new encoding to the fault localization technique based on error invariants and show that it produces more meaningful error explanations than previous approaches.

1 Introduction

Debugging programs is tedious. Often the most time consuming part in debugging is the task of *fault localization*. To localize the fault of an error, the programmer has to trace the program execution, e.g., starting from a malicious test input, and identify the *relevant* statements that explain the error. Using this information the programmer can then devise a solution to fix the program. Automated fault localization tools [1, 6–8, 11, 14, 15, 17, 19, 20] that reduce the manual effort involved in this task can significantly improve the programmer's work experience.

Particularly promising are proof-based fault localization techniques [6, 11]. These techniques rely on an encoding of error traces into so called *error trace formulas*. An error trace formula is an unsatisfiable logical formula. A proof of unsatisfiability of the error trace formula captures the reason why an execution of the error trace fails. By applying an automated theorem prover to obtain such proofs of unsatisfiability, the relevant statements for the error can be automatically identified. One advantage of this approach is that the proof of

⋆ Supported in part by the projects SAFEHR and COLAB, funded by Macau Science and Technology Development Fund.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 189–208, 2013.

unsatisfiability provides additional valuable information that can help the programmer understand the error, such as information about the program states at different points of execution.

A key limitation of existing proof-based fault localization techniques is that they only consider statements in the explanation of an error that have side-effects on the program's data. Information about the control flow of the program is ignored. This makes it difficult for the programmer to understand why the statements that are reported as relevant are actually reachable. Existing proof-based techniques therefore fail to explain common classes of program errors, such as faulty branching conditions.

In this paper, we propose a *flow-sensitive* proof-based fault localization technique. The result of a flow-sensitive fault localization not only explains why the error occurred, but also justifies why the statements leading to the error were executed. We give a basic algorithm for flow-sensitive fault localization based on our previous work on error invariants [6]. This algorithm applies fault localization recursively to explain the truth values of branching conditions along the error trace. We then observe that already the non-recursive algorithm from [6] can produce a flow-sensitive error explanation if one uses a simple modification in the encoding of the error trace formula. We refer to this new encoding of an error trace as the *flow-sensitive error trace formula*.

We discuss a number of examples that demonstrate the usefulness of flow-sensitive fault localization. We compare the results to the original fault localization based on error invariants and show that the new technique precisely explains why an error occurs.

Related Work. BugAssist [10,11] minimizes a given error trace obtained from bounded model checking using a Max-SAT algorithm. Similar to our previous work [6], BugAssist encodes the error in an unsatisfiable formula. The fault is localized by identifying fragments of the error trace that are not needed to prove unsatisfiability of the trace formula. This results in the limitation that only executable statements can be part of the minimized error trace. Branch conditions, and those statements that explain why they hold, are always omitted as the violation of the assertion causing the error can be proven even without considering them. The main contribution of our work is a new way to encode error traces as formulas such that branch conditions, and statements affecting them, can appear in the result of the localization.

A common approach to fault localization is to compare failing with successful executions (e.g., [1,7,8,14,15,17,19,20]). These approaches differ in the way the failing and successful executions are obtained, the way they compare executions, and in the information they report to the user. A detailed survey about the differences among these approaches can be found in [16,18]. The main difference to our approach is that we do not execute the program and that we do not need a successful execution.

In [17], an approach is presented that compares similar failing and successful executions of a program and returns a bug report that contains only those branch conditions that have been evaluated differently in both executions. From a given

```
1 void xZero(int input) {
2     int x = 1;
3     int y = input - 42;
4     if (y<0) {
5         x = 0;
6     }
7     //@ assert x != 0;
8 }
```

Fig. 1. The error occurs only if the `then` block is taken. The condition and the derivation of its truth value are important to reproduce the error.

```
1 int yZero(int input) {
2     int x = 0;
3     int y = input - 42;
4     if (y<0) {
5         y = 0;
6     }
7     //@ assert x != 0;
8 }
```

Fig. 2. The error occurs no matter which branch of the conditional is taken

failing execution of a program, they automatically construct a *similar* successful execution using a constraint solver. The evaluation of the branch conditions in both executions is recorded during execution, and the difference is reported to the user as a bug report. With this approach, we share the motivation that branch conditions not only provide valuable information for debugging but are often the reason for errors in a program and therefore are essential for fault localization. However, unlike the approach in [17], our approach reports not only branch conditions but also relevant statements, error invariants, and the variables that need to be tracked.

Similar to dynamic approaches there are static approaches that do fault localization by comparing feasible traces in the model of the program with counterexamples produced by a verifier. Ball et al. [1] present an algorithm to localize faults in counterexamples produced by the model checker SLAM. They isolate parts of a counterexample that do not occur on feasible traces. Groce et al. [7–9] use causal dependencies (see, e.g. [2]) and distance metrics for program executions to find minimal abstractions of error traces. In contrast to our approach, they use causal dependencies between variables and the difference between the failing and the successful execution to generate error reports.

2 Overview and Illustrative Example

We illustrate the benefits of flow-sensitive trace formulas on two simplified examples. Fig. 1 shows a procedure whose execution violates the assertion at line 7 if it is called with a value less than 42 for the parameter `input`. In this case, the assignment in line 5 is executed and the assigned value of `x` conflicts the assertion at line 7. An error is observable by executing an *error trace* starting from a state that satisfies an *error precondition*. In our example, the error trace is the sequence of statements obtained by restricting the program to the `then` branch of the conditional, i.e., including the statement in line 5. The error precondition is the formula `input < 42` (or any other formula that implies `input < 42`).

The problem of fault localization is to identify the statements in the error trace that are relevant for the error. Intuitively, a statement is relevant if the error no longer occurs after removing the statement from the trace. Various notions of relevancy have been considered in the literature depending on what it means to remove a statement from a trace.

In proof-based fault localization techniques [6, 11] relevancy of statements is determined by encoding the error into an unsatisfiable formula called the *extended trace formula*. This formula consists of a *trace formula* of the error trace in conjunction with the error precondition and the correctness assertion. A proof of unsatisfiability of the extended trace formula provides information about which statements are relevant. For our example, the extended trace formula is as follows:

$$(input < 42) \wedge (x = 1) \wedge (y = input - 42) \wedge (y < 0) \wedge \underline{(x' = 0)} \wedge \underline{(x' \neq 0)}.$$

The first conjunct is the error precondition, the last conjunct is the failing assertion, and the remaining conjuncts constitute the trace formula encoding of the error trace, e.g., the conjunct $(x' = 0)$ results from the statement in line 5. The conjuncts that are needed to prove the unsatisfiability of the formula are underlined, i.e., already $(x' = 0) \wedge (x' \neq 0)$ is contradictory. Thus, it appears as if only the statement in line 5 is relevant for the error. However, in order to reproduce the error we also need to know that `y<0` holds in line 4, otherwise the statement in line 5 will not be executed. Hence, we need the statement `y=input-42` and the precondition `input<42` to show that the condition in line 4 is true. Unfortunately, we cannot derive this information from the unsatisfiability proof because the values of `y` and `input` are irrelevant for the proof.

To overcome this problem, we introduce an alternative encoding of errors into so-called *flow-sensitive trace formulas*. A flow-sensitive trace formula keeps track of dependencies between statements and the branching conditions that are relevant for the reachability of these statements in the control flow graph of the program. A proof of unsatisfiability of such a formula includes a justification for the truth value of every branching condition on which a relevant statement depends. The flow-sensitive trace formula for the example in Fig. 1 is as follows:

$$\underline{(input < 42)} \wedge (x = 1) \wedge \underline{(y = input - 42)} \wedge \underline{(y < 0 \implies x' = 0)} \wedge \underline{(x' \neq 0)}$$

For a proof of unsatisfiability we still need the information that $x' = 0$ holds, but this information is now encoded in an implication $(y < 0 \implies x' = 0)$. The premise of the implication, $(y < 0)$, is the branching condition at line 4 that needs to be satisfied to reach the statement in line 5. The implication encodes that either the branching condition holds and the statement in line 5 is executed, or the branching condition does not hold and x' takes an arbitrary value (effectively over-approximating the `else` branch of the conditional). That is, either the implication $(y < 0 \implies x' = 0)$ is relevant for the proof or we can show that the value of x' is completely irrelevant. If the implication is relevant, then the branching condition $y < 0$ must also be relevant and so must be all the statements that affect its truth value. Hence, a statement that affects the

reachability of a relevant statement is also considered relevant. In this example, the unsatisfiability of the flow-sensitive trace formula can only be proven if we can show that the implication $(input < 42) \wedge (x = 1) \wedge (y = input - 42) \implies (y < 0)$ is valid. The conjunct $(y = input - 42)$ resulting from line 3, and the precondition $(input < 42)$, which are both part of the premise, are sufficient for this. Hence, using the flow-sensitive trace formula, we can still identify the conflict between line 5 and line 7, but in addition, we can also explain why line 5 is reached. In the end, only the statement in line 2 is considered irrelevant.

Flow-sensitive trace formulas also distinguish between conditional choices that are relevant to reach the error and those that are irrelevant. We illustrate this using the procedure `yZero` shown in Figure 2. The flow-sensitive trace formula of `yZero` is either

$$(input < 42) \wedge \underline{(x = 0)} \wedge (y = input - 42) \wedge (y < 0 \implies y' = 0) \wedge \underline{(x \neq 0)}$$

if the trace to the error goes through the if-statement at line 4, or

$$(input < 42) \wedge \underline{(x = 0)} \wedge (y = input - 42) \wedge (y \geq 0 \implies y' = y) \wedge \underline{(x \neq 0)}$$

if it does not. Note that both traces are error traces, as the correctness assertion $x \neq 0$ in line 7 is violated in each case. We reuse the error precondition $(input < 42)$ from the previous example, though we might as well use any other constraint on `input` (as long as it is satisfiable). To prove the formula unsatisfiable, it is sufficient to prove the contradiction between the conjunct $(x = 0)$, resulting from line 2, and the assertion $(x' \neq 0)$. The conjunct $(y < 0 \implies y' = 0)$, respectively $(y \geq 0 \implies y' = y.)$, is not needed. We thus conclude that the conditional choice in line 4 is irrelevant to reproduce the error. Similarly, we observe that the constraints resulting from the statement `y=input-42` and the precondition `input<42` are not needed. Hence, we further conclude that neither the value of `input`, nor the value of `y` are relevant to reproduce the error.

3 Preliminaries

We use first-order logic formulas to define programs. We assume standard syntax and semantics of such formulas and use $\top$ and $\bot$ to denote the Boolean constants for *true* and *false*, respectively. For a set of variables X, we denote by $\mathsf{Expr}(X)$ the set of terms built from variables in X and we denote by $\mathsf{Preds}(X)$ the set of all quantifier-free formulas with free variables in X.

Programs and Statements. Let X be a fixed set of variables, which we call *program variables*, and let $\mathcal{L}$ be a set of *labels*. A *program statement* st is either a conditional choice, a while loop, a sequence of statements, an assignment, or a label:

$$x \in X, \quad \ell \in \mathcal{L}, \quad e \in \mathsf{Expr}(X), \quad cond \in \mathsf{Preds}(X)$$
$$st ::= \texttt{if}\ cond\ \texttt{then}\ st\ \texttt{else}\ st \mid \texttt{while}\ cond\ \texttt{do}\ st$$
$$\mid st; st \mid x := e \mid \texttt{label}\ \ell$$

Labels have no operational meaning. They are only used to uniquely identify certain points during the execution of a program statement. We require therefore that each label ℓ occurs at most once in a statement. We use the short-hand notation $\ell : st$ for the program statement $\texttt{label}\ \ell; st$.

A program $\mathcal{P} = \langle st, \Phi \rangle$ consists of a program statement st, and an *assertion map* $\Phi : \mathcal{L} \to \mathsf{Preds}(X)$ which maps each label ℓ in st to a formula that should hold at the point of execution of st determined by ℓ.

Atomic Statements and Traces. We define the semantics of program statements and programs in terms of *atomic statements* (or simply statements), which are defined by the following grammar:

$$st^a ::= \texttt{if}\ cond \mid \texttt{endif} \mid x := e \mid \texttt{label}\ \ell \mid \texttt{havoc}\ cond$$

A *trace* π is a finite sequence of atomic statements. Let π be a trace of length n. For $0 \leq i < n$, we denote by $\pi[i]$ the i-th atomic statement of π, and for $0 \leq i < j < n$, we denote by $\pi[i, j]$ the sub-trace $\pi[i]; \ldots; \pi[j-1]$ of π. Traces obtained from programs do not contain $\texttt{havoc}\ cond$ statements. We will use such statements later to define abstract traces.

A program statement st defines a possibly infinite, prefix-closed set of traces $\mathit{Traces}(st)$. The set $\mathit{Traces}(st)$ is obtained by unwinding the loops in st arbitrarily often. Formally, we define the set of *complete traces* $\mathit{CTraces}(st)$ of a statement st as the least fixed point of the following system of equations:

$$\begin{aligned}
&\mathit{CTraces}(x := e) = \{x := e\}\\
&\mathit{CTraces}(\texttt{label}\ \ell) = \{\texttt{label}\ \ell\}\\
&\mathit{CTraces}(st_1\ ;\ st_2) = \{\, \pi_1; \pi_2 \mid \pi_1 \in \mathit{CTraces}(st_1), \pi_2 \in \mathit{CTraces}(st_2) \,\}\\
&\mathit{CTraces}(\texttt{if}\ cond\ \texttt{then}\ st_1\ \texttt{else}\ st_2) =\\
&\quad \{\, \texttt{if}\ cond; \pi; \texttt{endif} \mid \pi \in \mathit{CTraces}(st_1) \,\} \cup\\
&\quad \{\, \texttt{if}\ \neg cond; \pi; \texttt{endif} \mid \pi \in \mathit{CTraces}(st_2) \,\}\\
&\mathit{CTraces}(\texttt{while}\ cond\ \texttt{do}\ st) =\\
&\quad \{\texttt{if}\ \neg cond; \texttt{endif}\} \cup\\
&\quad \{\, \texttt{if}\ cond; \pi; \texttt{endif}; \pi' \mid \pi \in \mathit{CTraces}(st), \pi' \in \mathit{CTraces}(\texttt{while}\ cond\ \texttt{do}\ st) \,\}
\end{aligned}$$

The set of traces $\mathit{Traces}(st)$ of a program statement st is then defined as the set of all prefixes of its complete traces $\mathit{CTraces}(st)$. The traces of a program $\mathcal{P} = \langle st, \Phi \rangle$ are the traces of its program statement, i.e., $\mathit{Traces}(\mathcal{P}) = \mathit{Traces}(st)$.

The purpose of using the above notation for the syntax of programs and traces, instead of more common notations such as guarded commands and passive programs, is that it allows to identify the scope of a branching condition from the syntax. We might as well use a more common syntax but then we need to recompute this scope in a preprocessing step.

Semantics of Traces and Programs. A program *state* s is a function that assigns a value $s(x)$ to each program variable $x \in X$. We call the formulas $\mathsf{Preds}(X)$ state formulas and we write $s \models F$ to denote that a state s satisfies a state formula F.

For a variable $x \in X$ and $i \in \mathbb{N}$, we denote by $x^{\langle i \rangle}$ the variable obtained from x by adding i primes to it. The variable $x^{\langle i \rangle}$ models the value of x in a state that is shifted i time steps into the future. We extend this shift function from variables to sets of variables, as expected, and we denote by X' the set of variables $X^{\langle 1 \rangle}$. For a formula F with free variables from Y, we write $F^{\langle i \rangle}$ for the formula obtained by replacing each occurrence of a variable $y \in Y$ in F with the variable $y^{\langle i \rangle}$. We denote by $x^{\langle -i \rangle}$ the inverse operation of $x^{\langle i \rangle}$.

The formulas $\mathsf{Preds}(X \cup X')$ are called *transition formulas*. We use transition formulas to represent the semantics of statements in a trace, where the variables X' denote the values of the variables from X in the next state. We write $s, s' \models T$ to denote that the states s and s' satisfy the transition formula T, where s' is used to interpret the variables in X'. We associate with each atomic statement st^a a transition formula $T[st^a]$ as shown in Figure 3. Here, $\mathit{frame}(Y)$ denotes the formula $\bigwedge_{y \in Y} y = y'$. The atomic statement `havoc` *cond* assigns non-deterministic values to all variables in $\mathit{vars}(\mathit{cond})$ and has no effect on the values of the remaining variables. Note that these statements do not occur in traces from the program as our programs are deterministic.

st^a	$T[st^a]$
`if` *cond*	$\mathit{cond} \wedge \mathit{frame}(X)$
`endif`	$\mathit{frame}(X)$
`label` ℓ	$\mathit{frame}(X)$
$x := e$	$x' = e \wedge \mathit{frame}(X \setminus \{x\})$
`havoc` *cond*	$\mathit{cond}' \wedge \mathit{frame}(X \setminus \mathit{vars}(\mathit{cond}))$

Fig. 3. The transition formulas describing the semantics of the statements in traces

An *execution* of a trace π of length n is a sequence of states $s_0 \dots s_n$ such that for all $0 \leq i < n$, $s_i, s_{i+1} \models T[\pi[i]]$. We denote by $\mathit{Execs}(\pi)$ the set of all executions of π. The *trace formula* of π is the formula

$$\mathbf{TF}(\pi) := T[\pi[0]]^{\langle 0 \rangle} \wedge \ldots \wedge T[\pi[n-1]]^{\langle n-1 \rangle} \ .$$

The trace formula is satisfiable if and only if π has a (feasible) execution. That is, if $\mathit{Execs}(\pi)$ is non-empty. In fact, there is a one-to-one correspondence between the executions of π and the models of $\mathbf{TF}(\pi)$. We call a trace *feasible* if its trace formula is satisfiable, otherwise we call it *infeasible*.

A program $\mathcal{P} = (st, \Phi)$ is *safe* if for every trace $\pi \in \mathit{Traces}(\mathcal{P})$ whose final statement is a label statement `label` ℓ and every execution σ of π, the final state of σ satisfies $\Phi(\ell)$. If a program is not safe, an *error* can be witnessed along a trace. The error corresponds to a set of executions that violate the assertion associated with the last label of the trace.

Definition 1. *An* error *of a program $\mathcal{P} = \langle st, \Phi \rangle$ is a tuple (ψ, π, ϕ) where ψ and ϕ are state formulas and $\pi = st_0^a; \dots st_{n-2}^a;$ `label` ℓ is a trace of $\mathcal{P}$ with $\Phi(\ell) = \phi$ such that the following conditions hold:*

1. $\psi \wedge \mathbf{TF}(\pi)$ *is satisfiable, and*
2. $\psi \wedge \mathbf{TF}(\pi) \wedge \phi^{\langle n \rangle}$ *is unsatisfiable.*

That is, for an error (ψ, π, ϕ), no execution of the trace π that starts in a state satisfying ψ ends in a state satisfying the postcondition ϕ. However, there must exist at least one execution of π that starts in a state satisfying ψ. We refer to the unsatisfiable formula $\psi \wedge \mathbf{TF}(\pi) \wedge \phi^{\langle n \rangle}$ as the *extended trace formula* of the error.

4 Flow-Sensitive Fault Localization

Proof-based fault localization techniques such as the ones described in [11] and [6] use the unsatisfiability proof of the extended trace formula to identify the parts of the trace formula that are relevant for the error. However, the extended trace formula only encodes that the trace does not have any execution for the given pre and postcondition. That is, to show that this formula is unsatisfiable, it might be sufficient to identify a single statement in the trace that establishes a contradiction with the postcondition. Though, to understand why the execution is possible at all, and thus to understand why the undesired post-state is reachable on this execution, more statements might be necessary. Therefore we propose *flow-sensitive fault localization* which localizes the fault and further explains for each statement in the result of the localization why the conditions needed to reach this statement are satisfied.

In this section, we show how existing fault localization algorithms can be made flow-sensitive while using the underlying algorithm as a black box. We do this for the fault localization technique based on *error invariants* [6], but the same principle also applies to other algorithms.

Error Invariants. Given an error (ψ, π, ϕ) with trace π of length n, let $0 \leq i \leq n$ be a position in the trace[1]. An error invariant for position i is a state formula I such that (i) $\psi \wedge \mathbf{TF}(\pi[0, i]) \models I^{\langle i \rangle}$ and (ii) $I \wedge \mathbf{TF}\pi[i, n] \wedge \phi^{\langle n-i \rangle} \models \bot$. That is, I over-approximates the final states of the executions $\mathit{Execs}(\pi[0, i])$ that start in a state satisfying ψ. Furthermore, the final state of any execution of $\pi[i, n]$ that starts in a state satisfying the error invariant still violates ϕ. An error invariant is *inductive* for positions $i \leq j$ if it is an error invariant for both positions i and j.

Fault Localization. We formulate fault localization as the problem of finding an *abstract error* for a given error that describes the essence of why the error occurred. The abstract error comprises an abstract error trace that over-approximates the executions of the original error trace and fails for the same reason. We define these abstract error traces using inductive error invariants. If an error invariant I is inductive for positions $j > i$, then the statements between positions i and j are not needed to reproduce the error. We can drop

[1] Position n is the position where the assertion ϕ is supposed to hold.

these statements from the error trace and replace them by a *summary transition* that non-deterministically changes the values of variables, but preserves the error invariant I. The statements in the error trace for which we cannot find an encompassing inductive invariant is considered relevant and remains in the abstract error trace.

Definition 2 (Abstract error trace). *Given an error (ψ, π, ϕ) with trace π of length n. Let $\pi^{\#} = \texttt{havoc}\ I_0; st_1^a; \texttt{havoc}\ I_1; \ldots; st_k^a; \texttt{havoc}\ I_k$ be a trace where all I_i, $0 \leq i \leq k$, are state formulas. We call $\pi^{\#}$ an* abstract error trace *for (ψ, π, ϕ) if there exist positions $i_0 < \ldots < i_{k+1}$ such that $i_0 = 0$, $i_{k+1} = n + 1$, for all j with $1 \leq j \leq k$, $st_j^a = \pi[i_j]$, and for all j with $0 \leq j \leq k$, I_j is an inductive error invariant for positions i_j and $i_{j+1} - 1$. We call $(\psi, \pi^{\#}, \phi)$ an* abstract error *associated with error (ψ, π, ϕ).*

Example 3. Consider again the example program from Fig. 1. We get an error for $input = 41$ and the trace $x := 1;\ y := input - 42;\ \texttt{if}\ y < 0;\ x := 0;\ \texttt{endif};\ \texttt{label}\ \ell$ where the condition assigned to label ℓ is $x \neq 0$. One possible abstract error trace is $\texttt{havoc}\ \top;\ x := 0;\ \texttt{havoc}\ x = 0$.

Craig Interpolants. There is a close connection between error invariants and Craig interpolants that we exploit to compute error invariants using the extended trace formula associated with the error.

Given two formulas A and B whose conjunction is unsatisfiable, a Craig interpolant for (A, B) - hereafter called interpolant - is a formula I such that (a) $A \implies I$ is valid, (b) $B \wedge I$ is unsatisfiable, and (c) the free variables in I occur free in both A and B. This concept has been extended to *inductive sequences of interpolants* [13]. Given n formulas $F_1, \ldots, F_n$ whose conjunction is unsatisfiable, an inductive sequences of interpolants is a sequence $I_0, \ldots, I_n$ of $n + 1$ formula such that (a) I_0 is $\top$, (b) I_n is $\bot$, (c) $I_{i-1} \wedge F_i \implies I_i$ is valid for $0 < i \leq n$, and (d) the free variables in I_i occur free in both, the formulas with index less than or equal to i, and the remaining formulas. Such inductive sequences of interpolants can be computed automatically from proofs of unsatisfiability using an appropriate interpolation procedure (see, e.g., [12]). We assume that such an interpolation procedure is given.

For an error (ψ, π, ϕ) with trace π of length n and a position $0 \leq i \leq n$, let $A = \psi \wedge \mathbf{TF}(\pi[0, i])$ and $B = \mathbf{TF}(\pi[i, n])^{\langle i \rangle} \wedge \phi^{\langle n \rangle}$. Then for every interpolant I of (A, B), the formula $I^{\langle -i \rangle}$ is an error invariant for position i of π.

Flow-sensitivity. An abstract error trace $\pi^{\#}$ for an error (ψ, π, ϕ) is an abstraction of the concrete error trace π. Since the abstraction only has to preserve the error, it might lose information about the control flow that is vital to reproduce the error in the original program. For instance, the abstract error trace in Example 3 does not provide any information about how line 5, the assignment $x := 0$, is reached. In particular, the abstract error trace does not incorporate any information about the variable y that is used in the condition of the **if**-statement (line 4) surrounding this assignment. In general, an error might not be caused

directly by the execution of a specific statement, but by the reachability of that statement under the given error precondition. For short error traces it is often easy to see why a certain statement is reachable. However, for longer error traces this is non-trivial. Hence, automation in form of fault localization that precisely captures the relevant control flow is desirable.

We introduce *flow-sensitive* abstract error traces to solve this problem. A flow-sensitive abstract error trace $\pi^\#$ is an abstract error trace with the property that for every statement st^a in $\pi^\#$, a prefix of $\pi^\#$ can be used to explain why st^a is reachable in the original trace. To formalize this concept, we introduce two helper functions *Conds* and *Prev*. We denote by $Conds(\pi)$ the conditions needed to reach $\pi[n]$, i. e., the conditions of the **if**-statements whose corresponding `endif`-statements are not part of π:

$$\begin{aligned}
Conds(\pi) &= Conds(\emptyset, \pi, n) \\
Conds(S, \epsilon, 0) &= S \\
Conds(S, \pi \text{ ; } \texttt{endif}, i) &= Conds(S, \pi[0, Prev(\pi)], Prev(\pi)) \\
Conds(S, \pi \text{ ; } \texttt{if } cond, i) &= Conds(S \cup \{cond\}, \pi, i-1) \\
Conds(S, \pi \text{ ; } \texttt{label } \ell, i) &= Conds(S, \pi \text{ ; } x := e, i) \\
&= Conds(S, \pi \text{ ; } \texttt{havoc } cond, i) \\
&= Conds(S, \pi, i-1)
\end{aligned}$$

Here, ϵ denotes the empty trace. If $Conds(\pi)$ is used within a formula, it is interpreted as the conjunction of its elements. The helper function $Prev(\pi)$ computes for a trace of length n the position of the last **if**-statement that does not have a corresponding `endif`-statement in π:

$$\begin{aligned}
Prev(\pi) &= Prev(1, \pi, n) \\
Prev(0, \pi, i) &= i + 1 \\
Prev(k, \pi \text{ ; } \texttt{endif}, i) &= Prev(k+1, \pi, i-1) \\
Prev(k, \pi \text{ ; } \texttt{if } cond, i) &= Prev(k-1, \pi, i-1) \\
Prev(k, \pi \text{ ; } \texttt{label } \ell, i) &= Prev(k, \pi \text{ ; } \texttt{havoc } cond, i) \\
&= Prev(k, \pi \text{ ; } x := e, i) \\
&= Prev(k, \pi, i-1)
\end{aligned}$$

Definition 4 (Flow-sensitive). *Let P be a program. An abstract error $(\psi, \pi^\#, \phi)$ for an error (ψ, π, ϕ) of P is called* flow-sensitive *if for every statement st^a in $\pi^\#$ with $st^a = \pi[i] = \pi^\#[k]$, some prefix of $\pi^\#[0, k]$ is an abstract error trace for the error $(\psi, \pi[0, i], \neg Conds(\pi[0, i]))$ of the program that is obtained from P by inserting[2] the statement `label` ℓ before the statement $\pi[i]$ and mapping the fresh label ℓ to the assertion $\neg Conds(\pi[0, i])$.*

[2] Note that the insertion of labels into a program does not change the semantics of the program.

To make fault localization flow-sensitive, we need to know the scope of the statement, i.e., the conditions that have to hold for the statement to be reachable. With this knowledge, we can make any fault localization technique flow-sensitive by using recursive calls to find out why the conditions have to hold when reaching statements in conditional branches. As an example, we show in Algorithm 1 how to make the fault localization based on error invariants flow-sensitive. In general, this algorithm can easily be adapted to other fault localization techniques.

Algorithm 1. FSEI(ψ, π, ϕ): naive algorithm to compute error invariants for flow-sensitive fault localization.

Input: Pre-condition ψ, Trace π of length n, and post-condition ϕ
Output: Sequence of $n+1$ error invariants
$E_0, \ldots, E_n \leftarrow \mathit{ErrorInvariants}(\psi, \pi, \phi)$ (1)
$\mathit{Pos} \leftarrow \mathit{Changes}(E_0, \ldots, E_n)$ (2)
$\mathit{Conditions} \leftarrow \bigcup_{i \in \mathit{Pos}} \mathit{Scope}(i)$ (3)
foreach (i, cond) **in** $\mathit{Conditions}$ **do**
  $S_0, \ldots, S_i \leftarrow$ **FSEI**$(\psi, \pi[0,i], \neg \mathit{cond})$ (4)
  foreach $0 \leq j \leq i$ **do** $E_j \leftarrow E_j \wedge S_j$ (5)
return $E_0, \ldots, E_n$

In the line marked by (1) it uses an algorithm for fault localization based on error invariants [6] as a black box. This algorithm is supposed to return an error invariant for every position in the trace and one for the post-condition. Hence, for an input trace of length n the result consists of $n+1$ error invariants. The flow-sensitive localization then extracts, in (2), the positions in the sequence of error invariants where the invariant changes. These positions index the relevant statements in the trace. At this point, we get the result of the fault localization without considering flow-sensitivity. Hence, (1) and (2) could be substituted by any fault localization algorithm that computes a set of relevant statements for the error. From the list generated in (2) the algorithm extracts, in (3), the scope of every statement as a set of pairs of positions and conditions. Let j be the position of a statement in a trace π, then $\mathit{Scope}(j)$ returns the set of all pairs (i, cond) such that (1) $i < j$, (2) $\pi[i]$ is `if` cond, and (3) the corresponding `endif`-statement is not in $\pi[0, j]$.

When this information is available, we can reduce flow-sensitive fault localization to a simpler subproblem. For every pair (i, cond) we insert[2] a fresh label ℓ_i before the statement $\pi[i]$ and map the label to $\neg \mathit{cond}$. Then, in line (4), we recursively call the flow-sensitive fault localization for the error $(\psi, \pi[0,i], \neg \mathit{cond})$ in the modified program. Essentially we ask the question "Why does *cond* hold after executing $\pi[0,i]$ from states in ψ?"

To understand why this procedure works, we first note that the condition *cond* has to hold for all execution of $\pi[0,i]$ that start in ψ since our programs are deterministic. Hence, the tuple $(\psi, \pi[0,i], \neg \mathit{cond})$ is an error in the modified program.

In our algorithm, the recursive calls return error invariants that explain why every execution of the prefix up to the condition that starts in ψ must satisfy the condition. The results of the recursive calls have to be combined with the result of the initial call to the error invariant generation. We are only allowed to strengthen the error invariants. Otherwise we might introduce executions that do not violate the postcondition in which case the result would not be a flow-sensitive abstract error trace. Thus, we conjoin the invariants derived by the recursive call with the current invariants.

A binary search algorithm similar to the one presented in [6] can be used to find for each error invariant a maximal interval of positions for which this error invariant is inductive. We denote by **Localize** the procedure that takes as input a sequence of error invariants and an error (ψ, π, ϕ), and builds an abstract error $\mathbf{Localize}(\langle E_0, \ldots, E_n \rangle, (\psi, \pi, \phi))$, by replacing each subsequence $\pi[i, j]$ of π by `havoc` E_k, if E_k is an inductive invariant for positions $i < j$, for some $0 \leq k \leq n$.

Theorem 5. *Let* (ψ, π, ϕ) *be an error and* $E_0, \ldots, E_n = \mathbf{FSEI}(\psi, \pi, \phi)$ *a sequence of error invariants. Then,* $\mathbf{Localize}(\langle E_0, \ldots, E_n \rangle, (\psi, \pi, \phi))$ *is a flow-sensitive abstract error.*

A downside of Algorithm 1 is that it might need one recursive call to fault localization per **if**-statement in the trace, which results in a quadratic worst case complexity. This will be inefficient for long traces. We therefore propose a new encoding of an error into a *flow-sensitive trace formula*. The benefit of this new encoding is that it yields a more efficient non-recursive flow-sensitive fault localization algorithm.

5 Flow-Sensitive Trace Formulas

We denote the flow-sensitive trace formula of a trace π by $\mathbf{FSTF}(\pi)$. The idea behind flow-sensitive trace formulas is that, in addition to encoding the executions of the trace π, they also encode an over-approximation of the executions of all other traces that only differ from π in conditional statements. That is, if we need a statement that is only reachable under certain conditions to prove $(\psi \wedge \mathbf{FSTF}(\pi) \wedge \phi^{\langle n \rangle})$ unsatisfiable, we also prove that every execution starting in a state in ψ inevitably has to reach this condition.

Algorithm 2 shows how the flow-sensitive trace formula $\mathbf{FSTF}(\pi)$ is computed for a given trace π. Similar to a trace formula, we compute a conjunction of (appropriately shifted) transition formulas for each statement of π. However, instead of adding conjuncts for branch conditions `if` *cond*, we memorize the branch conditions that have to hold to reach a particular position on the trace (see ℓ_1 and ℓ_3), and for every statement other than `if` *cond* and `endif`, we add a chain of implications for these branch conditions to the transition formula of this statement (see ℓ_5). Hence, whenever we want to show that a statement is needed to prove that the trace formula contradicts with pre and postcondition, we also have to prove that all conditions needed to reach this statement can be

Algorithm 2. FSTF(π): flow-sensitive trace formula.

Input: Trace π of length n
Output: flow-sensitive trace formula ret

```
Stack conds;
Formula ret = true;
for (i = 0 to n − 1){
  if (π[i] is (if cond)) {
ℓ1   conds.push(cond^⟨i⟩);
ℓ2   ret = ret ∧ frame(X)^⟨i⟩;          //stub for TF(π[i])
  } else if (π[i] is (endif)) {
ℓ3   conds.pop();
ℓ4   ret = ret ∧ frame(X)^⟨i⟩;          //stub for TF(π[i])
  } else {
ℓ5   ret = ret ∧ ((⋀_{c∈conds} c) ⟹ (T[π[i]]^⟨i⟩));
  }
}
```

Algorithm 3. $FSErrInvs(\psi, \pi, \phi)$: algorithm to compute a flow-sensitive error invariants.

Input: Precondition ψ, Trace π of length n, and postcondition ϕ
Output: Sequence of $n+1$ error invariants
$I_0, \ldots, I_n \leftarrow InductiveInterpolants(\psi \wedge \mathbf{FSTF}(\pi) \wedge \phi)$ (1)
return $I_0^{\langle -0 \rangle}, \ldots, I_n^{\langle -n \rangle}$

satisfied. This guarantees that our fault localization will consider these branch conditions and all the statements affecting their truth values.

Algorithm 3 is our non-recursive flow-sensitive fault localization algorithm that computes the error invariants directly via Craig interpolation of the flow-sensitive trace formula. The resulting abstract error is still guaranteed to be flow-sensitive:

Theorem 6. *Let (ψ, π, ϕ) be an error with trace π of length n and $E_0, \ldots, E_n = FSErrInvs(\psi, \pi, \phi)$ be a sequence of error invariants computed by Algorithm 3. Then,* **Localize**$(\langle E_0, \ldots, E_n \rangle, (\psi, \pi, \phi))$ *is a flow-sensitive abstract error.*

Proof (sketch). Let the result of **Localize**$(\langle E_0, \ldots, E_n \rangle, (\psi, \pi, \phi))$ be $(\psi, \pi^\#, \phi)$ with $\pi^\# \equiv (\texttt{havoc}\ E_0, st_1^a; \texttt{havoc}\ E_1; \ldots; \texttt{havoc}\ E_k)$ where no st_i^a is a havoc statement. Since $\pi^\#$ is an abstract error trace by construction, we only have to show that it is also flow-sensitive. Note that **Localize** guarantees that for all $0 \leq i \leq k$ and $0 \leq j \leq k$, if $i \neq j$ then E_i is not an error invariant that can replace E_j.

Let j be a position in the abstract error trace with $0 < j \leq k$. Furthermore, let i_j denote the position of st_j^a in the concrete error trace π. If $Conds(\pi[0, i_j]) = \emptyset$,

i. e., the statement does not occur under an **if**-statement, flow-sensitivity holds trivially for this statement. Otherwise, we denote by ξ the formula corresponding to $\mathit{Conds}(\pi[0, i_j])$. Note that $\mathbf{FSTF}(\pi[i_j]) \equiv \xi \rightarrow \mathbf{TF}(\pi[i_j])$.

We know that E_j is an error invariant at position i_j and E_{j-1} is an error invariant at position $i_j - 1$. Furthermore, we know that the following equivalences hold:

$$\begin{aligned} \psi \wedge \mathbf{FSTF}(\pi[0, i_j]) &\equiv \psi \wedge \mathbf{FSTF}(\pi[0, i_j - 1]) \wedge \mathbf{FSTF}(\pi[i_j])^{\langle i_j - 1 \rangle} \\ &\equiv \psi \wedge \mathbf{FSTF}(\pi[0, i_j - 1]) \wedge \xi \rightarrow \mathbf{TF}(\pi[i_j])^{\langle i_j - 1 \rangle} \end{aligned}$$

If ξ does not hold in E_{j-1} it is not relevant for the error. Hence, the statement st_j^a cannot appear in the abstract error trace. This contradicts the property guaranteed by **Localize** considering E_j and E_{j-1}. Hence, a prefix of the abstract error trace up to the error invariant E_{j-1} has to ensure that ξ has to hold.

$\mathit{input} < 42$	
$x' = 1$	$\mathit{input} < 42$
$y' = \mathit{input} - 42$	$\mathit{input} < 42$
$y' < 0 \implies x'' = 0$	$y' < 0$
$x'' \neq 0$	$x'' = 0$

(a) using flow-sensitive trace formula

$\mathit{input} < 42$	
$x' = 1$	$\top$
$y' = \mathit{input} - 42$	$\top$
$y' < 0 \wedge x'' = 0$	$\top$
$x'' \neq 0$	$x'' = 0$

(b) using trace formula

Fig. 4. Interpolant derivation for the program from Figure 1

Example 7. Figure 4(a) shows the derived interpolants for the flow-sensitive trace formula of the error in our motivating example given in Fig. 1. Every conjunct of the trace formula is written on a single line. The first formula is the precondition of the error, the last formula is the postcondition, and the remaining formulas are the flow-sensitive trace formulas for the statements in the trace. The horizontal lines separating the different formula parts correspond to the positions where an interpolant is computed. Adjacent to each line, we annotate the computed interpolant. Note that each interpolant is unsatisfiable in conjunction with the formulas below the adjacent line[3]. Consider the last interpolant $x'' = 0$. This interpolant suffices to show that the postcondition is violated. The flow-sensitive encoding however forces the prover to justify why the condition $y' < 0$ holds. This justification is given in the preceding interpolants.

Figure 4(b) shows the derivation of interpolants for the same error encoded with the original extended trace formula. In this encoding, the fact $x'' = 0$ holds unconditionally. Therefore, only the last interpolant is non-trivial because no justification has to be given why the assignment statement $x = 0$ is actually reachable.

[3] The interpolants actually form an inductive sequence. We omit the initial $\top$ and final $\bot$ interpolant of this sequence.

```
1  void process() {
2    int x, y, z;
3    z = 0;
4    lock = 1;
5    if (x == 0) {
6      if (y == 0)
7        z = 1;
8    }
9    if (y != 0) {
10     z = y;
11   }
12   if (x != 0) {
13     z = 2;
14     lock = 0;
15   } else if (z > 0) {
16     z = 3;
17     lock = 0;
18   }
19   //@ assert lock == 0;
20 }
```

Fig. 5. Faulty locking with a simplified locking mechanism

6 Evaluation

For our prototype implementation we use the software model checker Kojak which is based on Ultimate [5] and the interpolating theorem prover SMTInterpol [4]. Kojak implements the fault localization algorithm based on error invariants [6]. We modified the implementation to perform flow-sensitive fault localization using the flow-sensitive trace formula encoding.

In the following, we demonstrate the benefits of flow-sensitive trace formulas in fault localization on two examples taken from the literature to which we have applied our implementation. We prefix statements in abstract error traces with their corresponding line numbers in the original program. For frame conditions we use the line number of the associated conditional choice and add the subscript *fc*. For the sake of presentation we limit ourselves to small examples. Compared to previous approaches, our new method results in longer abstract error traces since it now provides additional information about the program's control flow.

Faulty Locking. The first example is taken from [3] and shown in Figure 5. The program uses a locking mechanism to protect the access to variables. For the purpose of demonstration we simplified the lock/unlock steps by introducing a simple Boolean variable *lock*. The program sets $lock = 1$, then starts operations on the variables x, y, z, and finally checks if *lock* has been set to zero at the end of the procedure using an assertion in line 19. We use the model checker Kojak to check the safety of this assertion. The model checker finds a counterexample trace π and provides an initial failing state $\psi \equiv x = 0 \wedge y = -1$. Conjoining the obtained information yields the extended trace formula

$$\begin{aligned}(x = 0 \wedge y = -1) \wedge (z' = 0) \wedge \underline{(lock' = 1)} \wedge \\ ((x = 0) \wedge (y \neq 0) \wedge (z'' = z')) \wedge ((y \neq 0) \wedge (z''' = y)) \wedge \\ ((x = 0) \wedge (z''' \leq 0) \wedge \underline{(lock'' = lock')}) \wedge \underline{(lock'' = 0)} \ .\end{aligned}$$

We can see that only the variable *lock* is relevant to prove this formula unsatisfiable. Hence, the vanilla algorithm from [6] provides us only two inductive error

invariants, $\top$ and $lock = 1$. The corresponding abstract error trace is shown on the right side of Figure 6. The abstract error trace explains the direct cause of the error, but it does not show why the error is reachable.

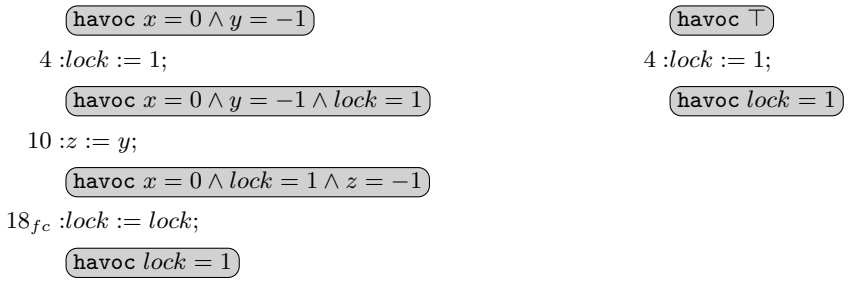


Fig. 6. Abstract error traces of the example in Fig. 5 with and without flow-sensitive encoding of the error.

Using flow-sensitive trace formulas for this error we get a different sequence of interpolants. Again, we show the formula one conjunct above the other with the interpolants written adjacent to the separating line:

$x = 0 \wedge y = -1$	$x = 0 \wedge y = -1$
$z' = 0$	$x = 0 \wedge y = -1$
$lock' = 1$	$x = 0 \wedge y = -1 \wedge lock' = 1$
$(x = 0) \Rightarrow (y \neq 0) \Rightarrow (z'' = z')$	$x = 0 \wedge y = -1 \wedge lock' = 1$
$(y \neq 0) \Rightarrow (z''' = y)$	$x = 0 \wedge lock' = 1 \wedge z''' = -1$
$(x = 0) \Rightarrow (z''' \leq 0) \Rightarrow (lock'' = lock')$	$lock'' = 1$
$lock'' = 0$	

From this sequence of interpolants we obtain the flow-sensitive abstract error trace shown in Figure 6. The first error invariant only represents the error precondition. Note that, in general, this invariant might be more general than the concrete values given by the model checker for the error precondition.

The second error invariant summarizes the control flow from line 5 to line 9 and the effect of the program up to line 5 that is relevant for the error. In particular, this invariant describes that the then-branch of the **if**-statement in line 5, the else-branch of the **if**-statement in line 6, and the then-branch of the **if**-statement in line 9 are taken. Furthermore, no assignment statement and no frame for the code between lines 5 to 9 affects the occurrence of the error.

The statement in line 10, however, has such an effect which can be seen in the next error invariant. The value of y becomes irrelevant for the remainder of the trace, but the value of the variable z now becomes relevant. From this error invariant it is easy to see that both conditions in lines 12 and 15 are not satisfied. Hence, the **if**-statement is skipped and we have to insert appropriate

frame conditions into our SSA-encoded error trace. This frame condition actually changes the error invariant. From this change we can conclude that the case distinction in lines 12 to 18 is incomplete. Note that this change is only enforced by the symbol condition for interpolants. To enforce this change in general, we introduce fresh auxiliary variables for the conditions needed to execute the statement. These variables are always local to exactly one interpolant and, hence, cannot be shifted. To improve readability we omitted these variables as they are not needed for the example.

```
1  int absValue(int input) {
2    int sign,abs;
3    abs = input;
4    if (input == 0)
5      return 0;
6    if (input<0) {
7      sign = -1;
8      printf("negative");
9    } else {
10     sign = 1;
11     printf("positive");
12   }
13   if(sign == -1) {
14     sign = input*-1;
15   } else {
16     abs = input;
17   }
18   //@ assert abs >= 0;
19   return abs;
20 }
```

Fig. 7. Example code of a faulty program that computes the absolute value and sign of the variable `input`

Faulty Absolute Value. The second example program is shown in Fig. 7. It computes the absolute value of an input variable *input*. The procedure `absValue` takes a variable `input` as input. If this variable is 0, the procedure returns without further computations (line 4-5). Otherwise the procedure sets `sign` to -1 if `input` is negative and to 1 if not (line 6-12) and prints a corresponding message to the console. Then, it computes the absolute value of `input` and writes it to `abs` (line 3-17). However, there is an error in the computation of the absolute value in line 14. The absolute value is written to `sign` instead of `abs`, which causes `abs` to have the original (negative) value of `input` that was assigned to it in line 3. This violates the assertion in line 18 which expects `abs` to be greater or equal to zero. The challenge here is that the error occurs because the variable `abs` is *not* modified in line 14. The above error is detected by Kojak and the error is witnessed using the example input `input=-1`. For this input, we can then extract an error trace π of the procedure `absValue`. The right column of Fig. 8 shows the abstract error trace obtained by analyzing π using the approach from [6] with non flow-sensitive encoding of trace formulas. To reproduce the failing execution it is sufficient to know that `input` is initially -1, and that in line 3 `abs` is set to the value of `input`. Since `abs` is not changed after that, these statements, together with the postcondition constitute an execution that fails the same way as the original error trace. However, what we are actually interested in is to see that the problem occurs because `sign` is modified instead of `abs` in line 14. This information is missing in the abstract error trace.

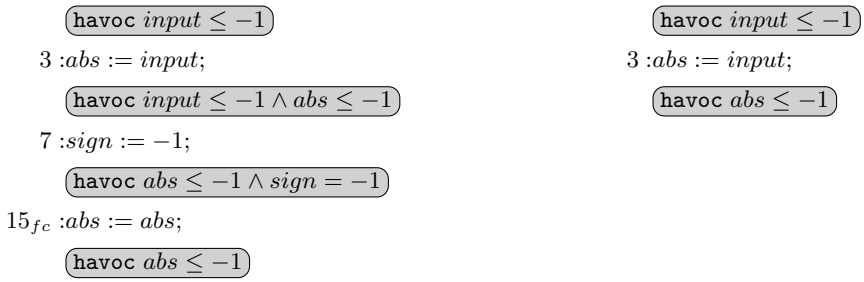


Fig. 8. Abstract error traces of the example in Fig. 7

We now consider the abstract error trace produced by the flow-sensitive algorithm. Again, we show the flow-sensitive trace formula and the sequence of interpolants computed for it:

$input = -1$	$input \leq -1$
$abs' = input$	$input \leq -1 \wedge abs' \leq -1$
$(input \neq 0) \Rightarrow \ldots$	$input =\leq 1 \wedge abs' \leq -1$
$(input < 0) \Rightarrow (sign' = -1)$	$abs' \leq -1 \wedge sign' = -1$
$(sign' = -1) \Rightarrow (sign'' = input * -1)$	$abs' \leq -1 \wedge sign' = -1$
$(sign' = -1) \Rightarrow (abs'' = abs')$	$abs'' \leq -1$
$abs'' \geq 0$	

From this sequence of interpolants we obtain the abstract error trace shown in Fig. 8.

The first error invariant in the abstract error trace is a generalization of the error precondition produced by KOJAK. It shows that the error also occurs if the value of `input` is any value less than or equal to -1. The next error invariant incorporates the effect of the assignment in line 3. Furthermore, it states that the **if**-statement in line 4 can be ignored when analyzing the error and that the then-branch of the **if**-statement in line 6 is important. The statement at line 7 which is contained in the then-branch is the next statement to change the error invariants. From the next error invariant we realize that, from line 7 to the end of the program, the value of the variable `input` is irrelevant, but the value of the variables `abs` and `sign` should be tracked. Furthermore, this invariant states that the then-branch of the **if**-statement in line 13 is taken. In this branch, the variable `abs` is not assigned, but the else-branch assigns it. Hence, we get a frame condition which then changes the error invariant again. From this change we can see that the missing assignment to `abs` is a potential reason for the failing assertion.

7 Conclusion

We have introduced the concept of flow-sensitive trace formulas. This new encoding of error traces into logical formulas enables proof-based fault localization methods to explain why relevant statements are reached in an erroneous execution of a program. Flow-sensitive trace formulas encode conditional choices of the error trace in such a way that the theorem prover can argue about the validity of the condition in the context of its prefix on the error trace. The resulting proof of unsatisfiability provides control-flow-related information on why the error occurred. We applied the flow-sensitive trace formula encoding to our fault localization technique based on error invariants. The resulting method identifies irrelevant portions of the code and finds a justification for the reachability of the remaining portions. Our evaluation shows that, while the produced abstract error traces are longer, they provide more useful error explanations. We therefore believe that the new encoding helps the programmer considerably to understand the faulty code fragment.

The application of the flow-sensitive trace formula encoding is not restricted to fault locations based on error invariants but can also be used in other methods. We will study such applications to other methods in our future work.

References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. SIGPLAN Not., 97–105 (2003)
2. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining Counterexamples Using Causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009)
3. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. SIGSOFT Softw. Eng. Notes 29(6), 73–82 (2004)
4. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
5. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via Interpolants. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 186–201. Springer, Heidelberg (2012)
6. Ermis, E., Schäf, M., Wies, T.: Error Invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012)
7. Groce, A.: Error Explanation with Distance Metrics. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 108–122. Springer, Heidelberg (2004)
8. Groce, A., Kroening, D.: Making the Most of BMC Counterexamples. ENTCS, 67–81 (2005)
9. Groce, A., Kroning, D., Lerda, F.: Understanding Counterexamples with `explain`. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004)
10. Jose, M., Majumdar, R.: Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 504–509. Springer, Heidelberg (2011)

11. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI 2011, pp. 437–446. ACM (2011)
12. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci., 101–121 (2005)
13. 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)
14. Qi, D., Roychoudhury, A., Liang, Z., Vaswani, K.: Darwin: an approach for debugging evolving programs. In: ESEC/SIGSOFT FSE, pp. 33–42 (2009)
15. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39 (2003)
16. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
17. Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: ASE, pp. 347–351. ACM (2005)
18. Wong, W.E., Debroy, V.: Software fault localization (2009)
19. Zeller, A.: Isolating cause-effect chains from computer programs. In: SIGSOFT FSE, pp. 1–10 (2002)
20. Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: ICSE, pp. 272–281. ACM, New York (2006)

Static Analysis of String Encoders and Decoders*

Loris D'Antoni[1] and Margus Veanes[2]

[1] University of Pennsylvania
lorisdan@cis.upenn.edu
[2] Microsoft Research
margus@microsoft.com

Abstract. There has been significant interest in static analysis of programs that manipulate strings, in particular in the context of web security. Many types of security vulnerabilities are exposed through flaws in programs such as string encoders, decoders, and sanitizers. Recent work has focused on combining automata and satisfiability modulo theories techniques to address security issues in those programs. These techniques scale to larger alphabets such as Unicode, that is a de facto character encoding standard used in web software.

One approach has been to use character predicates to generalize finite state transducers. This technique has made it possible to perform precise analysis of a large class of typical sanitization routines. However, it has not been able to cope well with decoders, that often require to read more than one character at a time. In order to overcome this limitation we introduce a conservative generalization of Symbolic Finite Transducers (SFTs) called Extended Symbolic Finite Transducers (ESFTs) that incorporates the notion of a bounded lookahead. We demonstrate the advantage ESFTs on analyzing programs for which previous approaches did not scale.

In our evaluation we use a UTF-16 to UTF-8 translator (*utf8encoder*) and a UTF-8 to UTF-16 translator (*utf8decoder*). We show, among other properties, that *utf8encoder* and *utf8decoder* are functionally correct.

1 Introduction

There has been significant recent interest in decision procedures for solving string constraints. Much of this work has focused on designing domain specific decision procedures for string analysis that use state-of-the art constraint solvers in the backend [17,4,19,20]. Many of the tools use automata based techniques, including JSA [5], and Bek [9]. A comprehensive comparison of various algorithmic design choices in this space is studied in [10]. The growing interest in string analysis has also started a discussion for developing standards for regular expressions modulo alphabet theories [3] to unify some of the notations and underlying theory in the tool support.

* This work was done during an internship at Microsoft Research and this research was partially supported by NSF Expeditions in Computing award CCF 1138996.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 209–228, 2013.

One reason for this focus is *security vulnerabilities* caused by strings. Some recent work has studied sanitizer correctness through static analysis based on automata theory [12,5,13], including the Bek project and symbolic transducers [9] that our work is based on. Here we extend the analysis of Bek to a richer, more expressive class of problems. In particular we consider string coders that require symbolic *lookahead*. Symbolic lookahead allows programs to read more than one symbol at a time. For example, in order to decode a (html encoded) string `"&#38;"` back to the string `"&"` a lookahead of two digits is needed. Concretely, in the paper we consider unicode encodings UTF-16 and UTF-8, that have emerged as the most commonly used character encodings. UTF-8 is used for representing Unicode text in text files and is perhaps the most widely accepted character encoding standard in the internet today. UTF-16 is used for in-memory representation of characters in modern programming and scripting languages. Transformations between these two encodings are ubiquitous.

Despite the wide adoption of these encodings, their analysis is difficult, and carefully crafted *invalid* UTF-8 sequences have been used to bypass security validations. Several attacks have been demonstrated [16] based on over-encoding the characters '`.`' and '`/`' in malformed URLs. For example, the invalid sequence $[\texttt{C0}_{16}, \texttt{AF}_{16}]$ (that decodes to '`/`') has been used to bypass a literal check in the Microsoft IIS server (in unpatched Windows 2000 SP1) to determine if a URL contains "`../../`" by encoding it as "`..%C0%AF../`". Similar vulnerability exists in Apache Tomcat ($\leq$ 6.0.18), where "`%C0%AE`" has been used for encoding '`.`' [14]. Further attacks use double-encoding [15]. We show how our new extension of symbolic transducers can make analysis of such coding routines possible.

Our analysis starts from a compilation from Bek programs to *symbolic transducers* (ST). In a symbolic transducer, transitions are annotated with logical *formulas* instead of specific characters, and the transducer takes the transition on any input character that satisfies the formula. A symbolic transducer is then transformed to a representation called *extended symbolic finite transducer* (ESFT), that uses lookahead to avoid state space explosion. For example, an ESFT may treat the pattern `"&#[0-9]{6};"` of an html decoder using a *single* transition rather than *100k* transitions required by an SFT (without lookahead). Our representation enables leveraging *satisfiability modulo theories (SMT) solvers*, tools that take a formula and attempt to find inputs satisfying that formula. These solvers have become robust in the last several years and are used to solve complicated formulas in a variety of contexts. At the same time, our representation allows leveraging automata theoretic methods to reason about strings of unbounded length, which is not possible via direct encoding to SMT formulas. One advantage of SMT solvers is that they work with formulas from any theory supported by the solver, while other previous approaches are specialized to specific types of inputs. This is a crucial feature for our algorithms and analysis, in particular we use a combination of theories, involving sequences, numbers, and records.

After the analysis, programs written in Bek can be compiled back to traditional languages such as JavaScript or C#. This ensures that the code analyzed

is functionally equivalent to the code which is actually deployed for sanitization, up to bugs in our compilation. Bek is available online [2].

This paper makes the following contributions:

- it introduces ESFTs as a new effective model for analysis of string coders;
- it presents an algorithm for STs register elimination that improves the efficiency and expressiveness of the previous state of the art technique based on exhaustive exploration;
- it proves *UTF8* encoder and decoder to be correct; and
- it uses realistic coding routines to show how ESFTs scale for big programs.

We first define ESFTs (Section 2) and STs with registers (Section 2.2). Secondly, we provide an algorithm to transform a subclass of STs into ESFTs (Section 3). We then describe UTF-8 encoders and decoders and their Bek implementation (Section 4). We use our technique to prove those coders correct. Finally, we show how our technique scales for bigger programs (Section 5).

2 Extended Symbolic Finite Transducers

We assume a *background structure* that has a recursively enumerable (r.e.) multi-typed carrier set or *background universe* $\mathcal{U}$, and is equipped with a language of function and relation symbols with fixed interpretations. We use τ, σ and γ to denote types, and we write $\mathcal{U}^\tau$ for the corresponding sub-universe of elements of type τ. As a convention, we abbreviate $\mathcal{U}^\sigma$ by Σ and $\mathcal{U}^\gamma$ by Γ, due to their frequent use. The Boolean type is $\mathbb{B}$, with $\mathcal{U}^\mathbb{B} = \{\mathfrak{t}, \mathfrak{f}\}$ and the integer type is $\mathbb{Z}$. Terms and formulas are defined by induction over the background language and are assumed to be well-typed. The type τ of a term t is indicated by $t : \tau$. Terms of type $\mathbb{B}$, or Boolean terms, are treated as formulas, i.e., no distinction is made between formulas and Boolean terms. A *k-tuple type* is a type $\mathbb{T}\langle\tau_0, \ldots, \tau_{k-1}\rangle$ where $k \geq 0$ and all τ_i are types. The 0-tuple type $\mathbb{T}\langle\rangle$ is assumed to be such that $\mathcal{U}^{\mathbb{T}\langle\rangle}$ is the singleton sub-universe $\{()\}$ and the 1-tuple type $\mathbb{T}\langle\tau\rangle \stackrel{\text{def}}{=} \tau$. If τ is a type and $k \geq 0$ then τ^k stands for the type $\mathbb{T}\langle\tau_0, \ldots, \tau_{k-1}\rangle$ of k-way Cartesian product where all $\tau_i = \tau$. For example, $\mathbb{Z}^2$ is $\mathbb{T}\langle\mathbb{Z}, \mathbb{Z}\rangle$. If t is a k-tuple ($k > 1$) then $\pi_i(t)$, also written $t[i]$, projects the i'th element of t for $0 \leq i < k$. The k-tuple constructor for $k > 1$ is simply $(t_0, \ldots, t_{k-1})$.

If τ is a type, then τ^* is the type over finite sequences of elements of type τ. We assume the standard accessors $\mathit{head} : \tau^* \to \tau$ and $\mathit{tail} : \tau^* \to \tau^*$ over sequences and the constructors $\mathit{cons} : \tau \times \tau^* \to \tau^*$ and $[] : \tau^*$. A term $\mathit{cons}(t_0, \mathit{cons}(t_1, \ldots, \mathit{cons}(t_{n-1}, [])))$ of sort τ^* is also denoted by $[t_0, t_1, \ldots, t_{n-1}]$ and is called an *explicit sequence of length n*. We use the following shorthands to access elements of a sequence $t : \tau^*$, $\mathit{tail}^0(t) \stackrel{\text{def}}{=} t$, $\mathit{tail}^{k+1}(t) \stackrel{\text{def}}{=} \mathit{tail}(\mathit{tail}^k(t))$, and for $k \geq 0$, $t[k] \stackrel{\text{def}}{=} \mathit{head}(\mathit{tail}^k(t))$. Given a set S, we write S^* for the *Kleene closure* of S. The justification behind overloading the $*$-operator both as a type annotator and Kleene closure operator is that, for any type τ, we assume $\mathcal{U}^{(\tau^*)} = (\mathcal{U}^\tau)^*$. In particular $\mathcal{U}^{(\sigma^*)} = \Sigma^*$ and $\mathcal{U}^{(\gamma^*)} = \Gamma^*$.

All elements in $\mathcal{U}$ are also used as constant terms. A term without free variables (such as a constant term) is *closed*. Closed terms t have standard Tarski semantics $[\![t]\!]$ over the background structure. Substitution of a variable $x:\tau$ in t by a term $u:\tau$ is denoted by $t[x/u]$.

A *λ-term* is an expression of the form $\lambda\bar{x}.t$, where $\bar{x}$ is a (possibly empty) tuple of distinct variables, and t is a term all of whose free variables occur in $\bar{x}$. It is sometimes technically convenient to view $\bar{x}$ as a single variable of the corresponding product (tuple) type.

To indicate the types, we say *$(\sigma \to \gamma)$-term* for a λ-term $\lambda x.t$ such that $x:\sigma$ and $t:\gamma$. A $(\sigma \to \gamma)$-term $f = \lambda x.t$ denotes the function $[\![f]\!]$ that maps $a \in \Sigma$ to $[\![t[x/a]]\!] \in \Gamma$. We use f, g, h to stand for λ-terms. We do not distinguish between the λ-term $\lambda().t$ and t.

A $(\sigma \to \mathbb{B})$-term is called a *σ-predicate*. We use φ and ψ for σ-predicates and, for $a \in \Sigma$, we write $a \in [\![\varphi]\!]$ for $[\![\varphi]\!](a) = \mathtt{t}$. Given a $(\sigma \to \gamma)$-term $f = (\lambda x.t)$ and a term $u:\sigma$, $f(u)$ stands for the term $t[x/u]$. A σ-predicate φ is *unsatisfiable* when $[\![\varphi]\!] = \emptyset$; φ is *satisfiable*, otherwise. A $(\sigma \to \gamma^*)$-term $f = \lambda x.[t_0, \ldots, t_{n-1}]$ is called a *$(\sigma \to \gamma)$-sequence* and $|f| \stackrel{\text{def}}{=} n$.

A *label theory* is given by an r.e. set Ψ of formulas that is closed under Boolean operations, substitution, equality and if-then-else terms. When talking about satisfiability of formulas, we assume implicit λ-closures. A label theory Ψ is *decidable* when satisfiability for $\varphi \in \Psi$, $IsSat(\varphi)$, is decidable.

Next, we describe an extension of finite state transducers through a symbolic representation of labels and by adding a lookahead component to the rules.

Definition 1. An *Extended Symbolic Finite Transducer (ESFT)* over $\sigma \to \gamma$ is a tuple $A = (Q, q^0, R)$,

- Q is a finite set of *states*;
- $q^0 \in Q$ is the *initial state*;
- R is a finite set of *rules*, $R = \Delta \cup F$, where
- Δ is a set of *transitions* $r = (p, \ell, \varphi, f, q)$, denoted $p \xrightarrow[\ell]{\varphi/f} q$, where
  - $p \in Q$ is the *start* state of r;
  - $\ell \geq 1$ is the *lookahead* of r;
  - φ, the *guard* of r, is a σ^ℓ-predicate;
  - f, the *output* of r, is a $(\sigma^\ell \to \gamma)$-sequence;
  - $q \in Q$ is the *end* state of r.
- F is a set of *final rules* $r = (p, \ell, \varphi, f)$, denoted $p \xrightarrow[\ell]{\varphi/f} \bullet$, with components as above and where ℓ is allowed to be 0.

The *lookahead* of A is the maximum of all lookaheads of rules in R.

We use the following abbreviated notation for rules, by omitting explicit λ's. We write

$$p \xrightarrow[\ell]{t/[u_0,\ldots,u_k]} q \quad \text{for} \quad p \xrightarrow[\ell]{\lambda\bar{x}.t/\lambda\bar{x}.[u_0,\ldots,u_k]} q,$$

where t and u_i are terms whose free variables are among $\bar{x} = (x_0, \ldots, x_{\ell-1})$. Final rules are a generalization of final states. A final rule with lookahead ℓ applies

only when the remaining input has *exactly* ℓ elements remaining as opposed to a transition with lookahead ℓ that applies when the remaining input has *at least* ℓ elements remaining.

The typical case of a final rule that corresponds to the classical final state p is $p \xrightarrow[0]{\mathtt{t}/[]} \bullet$, i.e., p accepts the empty input and produces no additional outputs. But there could be a non-empty output like in $p \xrightarrow[0]{\mathtt{t}/[\text{`\#'}]} \bullet$. There could also be a final rule with a non-zero lookahead. For example, suppose that characters are integers, the state is q, and there are two rules from q, a final rule: if there is a single input character remaining it is output "as is" $q \xrightarrow[1]{\mathtt{t}/[x_0]} \bullet$; and a transition, if there are at least two input characters, their sum is output $q \xrightarrow[2]{\mathtt{t}/[x_0+x_1]} q$. It is not possible to separate these two cases without introducing nondeterminism, if final rules with positive lookahead are not allowed. It is also not practical to lift the input type by adding a new "end of input" symbol as is done in the classical case. Such type lifting non trivially affects properties of the label theory and complicates use of specific theories over a given type such as standard linear arithmetic.

An ESFT with lookahead 1 and whose all final rules have lookahead 0 is an SFT [20]. In the sequel let $A = (Q, q^0, R)$, $R = \Delta \cup F$, be a fixed ESFT over $\sigma \to \gamma$. The semantics of rules in R is as follows:

$$[\![p \xrightarrow[\ell]{\varphi/f} q]\!] \stackrel{\text{def}}{=} \{p \xrightarrow{[a_0,\ldots,a_{\ell-1}]/[\![f]\!](a_0,\ldots,a_{\ell-1})} q \mid (a_0,\ldots,a_{\ell-1}) \in [\![\varphi]\!]\}$$

We write $s_1 \cdot s_2$ for concatenation of two sequences s_1 and s_2.

Definition 2. For $\boldsymbol{a} \in \Sigma^*, \boldsymbol{b} \in \Gamma^*, q \in Q, q' \in Q \cup \{\bullet\}$, define $q \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q'$ as follows: there exists $n \geq 0$ and $\{p_i \xrightarrow{\boldsymbol{a}_i/\boldsymbol{b}_i} p_{i+1} \mid i \leq n\} \subseteq [\![R]\!]$ such that

$$\boldsymbol{a} = \boldsymbol{a}_0 \cdot \boldsymbol{a}_1 \cdots \boldsymbol{a}_n, \quad \boldsymbol{b} = \boldsymbol{b}_0 \cdot \boldsymbol{b}_1 \cdots \boldsymbol{b}_n, \quad q = p_0, \quad q' = p_{n+1}.$$

Let also $q \xrightarrow{[]/[]}\!\!\!\!\rightarrow_A q$ for all $q \in Q$.

Does lookahead add expressiveness compared to SFTs? For finite Σ, the answer is *no*, because any concrete transition $p \xrightarrow{[a_0,a_1]/\boldsymbol{b}} q$ can be split into two transitions $p \xrightarrow{[a_0]/\boldsymbol{b}} p' \xrightarrow{[a_1]/[]} q$ where p' is a new (non final) state (as a consequence of the standard form [22, Theorem 2.17]). However, ESFTs are strictly more expressive than SFTs as the following example clearly illustrates. In general, ESFTs with lookahead $k+1$ are strictly more expressive than ESFTs with lookahead k.

Example 1. Let A be following ESFT

$$A = (\{q\}, q, \{q \xrightarrow[2]{(x_0=x_1)/[]} q, q \xrightarrow[0]{\mathtt{t}/[]} \bullet\})$$

Then $q \xrightarrow{\boldsymbol{a}/[]}\!\!\!\!\rightarrow_A \bullet$ iff $\boldsymbol{a}[2*i] = \boldsymbol{a}[2*i+1]$ for all $i \geq 0$. No SFT can express this dependency. ⊠

The above example can be generalized to any k. For a function $\mathbf{f} : X \to 2^Y$, define the *domain of* $\mathbf{f}$ as $\mathscr{D}(\mathbf{f}) \stackrel{\text{def}}{=} \{x \in X \mid \mathbf{f}(x) \neq \emptyset\}$; $\mathbf{f}$ is *total* when $\mathscr{D}(\mathbf{f}) = X$.

Definition 3. The *transduction of* A, $\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \mid q^0 \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A \bullet\}$.

The following subclass of SFTs captures transductions that behave as partial functions from Σ^* to Γ^*.

Definition 4. A is *single-valued* when $|\mathscr{T}_A(\boldsymbol{a})| \leq 1$ for all $\boldsymbol{a} \in \Sigma^*$.

A sufficient condition for single-valuedness is determinism. We define $\varphi \curlywedge \psi$, where φ is a σ^m-predicate and ψ a σ^n-predicate, as the $\sigma^{\max(m,n)}$-predicate $\lambda(x_1, \ldots, x_{\max(m,n)}).\varphi(x_1, \ldots, x_m) \wedge \psi(x_1, \ldots, x_n)$. We define *equivalence of* f *and* g *modulo* φ, $f \equiv_\varphi g$, as: $\mathit{IsValid}(\lambda \bar{x}.(\varphi(\bar{x}) \Rightarrow f(\bar{x}) = g(\bar{x})))$.

Definition 5. A is *deterministic* if and only if for all $p \xrightarrow[\ell]{\varphi/f} q, p \xrightarrow[\ell']{\varphi'/f'} q' \in R$ the following holds:

(a) Assume $q, q' \in Q$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $q = q'$, $\ell = \ell'$ and $f \equiv_{\varphi \curlywedge \varphi'} f'$.
(b) Assume $q = q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ and $\ell = \ell'$ then $f \equiv_{\varphi \curlywedge \varphi'} f'$.
(c) Assume $q \in Q$ and $q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $\ell > \ell'$.

Proposition 1. *If A is deterministic then A is single-valued.*

Determinism is not a necessary condition for single-valuedness. Moreover, deterministic ESFTs with lookahead $k+1$ are in general more expressive and more succinct than deterministic ESFTs with lookahead k. A few examples of ESFTs are given below to illustrate the definitions.

When A is total and single valued, and $\boldsymbol{a} \in \mathscr{D}(A)$, we write $A(\boldsymbol{a})$ for the value $\boldsymbol{b}$ such that $\mathscr{T}_A(\boldsymbol{a}) = \{\boldsymbol{b}\}$. In other words, we treat A as a function from Σ^* to Γ^*.

Example 2. Consider characters as their integer codes. We construct an ESFT *Decode* over $\mathbb{Z} \to \mathbb{Z}$ that replaces all occurrences of the regex pattern `#[0-9][0-9]` in the input with the corresponding encoded character. For example, since the code `'A'` $= 65$ and `'B'` $= 66$, we have *Decode*(`"##65#66#"`) $=$ `"#AB#"`.[1]

The states are q_0 and q_1, where q_0 is the initial state. The intuition behind the final rules is the following. In q_0 there is no unfinished pattern so the output is $[]$, while in q_1 the symbol `'#'` is the prefix of the unfinished pattern that needs to be output upon reaching the end of the input, and if there is only a single character x_0 remaining in the input then the output is [`'#'`, x_0].

[1] A literal string `"ABC"` stands for the sequence [`'A'`, `'B'`, `'C'`], where `'A'` is the character code of letter `A`.

The rules of D are as follows where *IsDigit* is the predicate $\lambda x.\texttt{`0'} \leq x \leq \texttt{`9'}$ (recall that $\texttt{`0'} = 48$ and $\texttt{`9'} = 57$).

$$F = \{\ q_0 \xrightarrow[0]{\mathrm{t}/[]} \bullet,\quad q_1 \xrightarrow[0]{\mathrm{t}/[\texttt{`\#'}]} \bullet,\quad q_1 \xrightarrow[1]{\mathit{IsDigit}(x_0)/[\texttt{`\#'},x_0]} \bullet\}$$

$$\begin{aligned}\Delta = \{\ & q_0 \xrightarrow[1]{(x_0 \neq \texttt{`\#'})/[x_0]} q_0,\quad q_0 \xrightarrow[1]{(x_0 = \texttt{`\#'})/[]} q_1,\quad q_1 \xrightarrow[1]{(x_0 = \texttt{`\#'})/[\texttt{`\#'}]} q_1,\\ & q_1 \xrightarrow[1]{(x_0 \neq \texttt{`\#'} \wedge \neg \mathit{IsDigit}(x_0))/[\texttt{`\#'},x_0]} q_0,\\ & q_1 \xrightarrow[2]{(\mathit{IsDigit}(x_0) \wedge x_1 \neq \texttt{`\#'} \wedge \neg \mathit{IsDigit}(x_1))/[\texttt{`\#'},x_0,x_1]} q_0,\\ & q_1 \xrightarrow[2]{(\mathit{IsDigit}(x_0) \wedge x_1 = \texttt{`\#'})/[\texttt{`\#'},x_0]} q_1,\\ & q_1 \xrightarrow[2]{(\mathit{IsDigit}(x_0) \wedge \mathit{IsDigit}(x_1))/[10*(x_0-48)+x_1-48]} q_0\}\end{aligned}$$

The last three rules have non-overlapping guards because the conditions on x_1 are mutually exclusive. An equivalent SFT would require a state p_d for each $d \in [\![\mathit{IsDigit}]\!]$ and a rule $q_1 \xrightarrow[1]{x_0 = d/[]} p_d$ in order to eliminate the rules with lookahead 2. ⊠

2.1 Composition of ESFTs

For composing ESFTs we first convert them to STs, as explained in Section 2.2, and then convert the resulting ST back to an ESFT using the semi-decision procedure explained in Section 3. In general, ESFTs are not closed under composition, as shown next.

Given $\mathbf{f} : X \to 2^Y$ and $\mathbf{x} \subseteq X$, $\mathbf{f}(\mathbf{x}) \stackrel{\text{def}}{=} \bigcup_{x \in \mathbf{x}} \mathbf{f}(x)$. Given $\mathbf{f} : X \to 2^Y$ and $\mathbf{g} : Y \to 2^Z$, $\mathbf{f} \circ \mathbf{g}(x) \stackrel{\text{def}}{=} \mathbf{g}(\mathbf{f}(x))$. This definition follows the convention in [7], i.e., $\circ$ applies first $\mathbf{f}$, then $\mathbf{g}$, contrary to how $\circ$ is used for standard function composition. The intuition is that $\mathbf{f}$ corresponds to the relation $R_{\mathbf{f}} : X \times Y$, $R_{\mathbf{f}} \stackrel{\text{def}}{=} \{(x,y) \mid y \in \mathbf{f}(x)\}$, so that $\mathbf{f} \circ \mathbf{g}$ corresponds to the binary relation composition $R_{\mathbf{f}} \circ R_{\mathbf{g}} \stackrel{\text{def}}{=} \{(x,z) \mid \exists y(R_{\mathbf{f}}(x,y) \wedge R_{\mathbf{g}}(y,z))\}$.

Definition 6. *A class of transducer C is closed under composition iff for every $\mathscr{T}_1$ and $\mathscr{T}_2$ that are C-definable $\mathscr{T}_1 \circ \mathscr{T}_2$ is also C-definable.*

Theorem 1. *ESFTs are not closed under composition.*

Proof. We show two ESFTs whose composition cannot be expressed by any ESFT. Let A be following ESFT over $\mathbb{Z} \to \mathbb{Z}$

$$A = (\{q\}, q, \{q \xrightarrow[2]{\mathrm{t}/[x_1,x_0]} q, q \xrightarrow[0]{\mathrm{t}/[]} \bullet\}).$$

and B be following ESFT over $\mathbb{Z} \to \mathbb{Z}$

$$B = (\{q_0, q_1\}, q_0, \{q_0 \xrightarrow[1]{\mathrm{t}/[x_0]} q_1, q_1 \xrightarrow[2]{\mathrm{t}/[x_1,x_0]} q_1, q_1 \xrightarrow[1]{\mathrm{t}/[x_0]} \bullet\})$$

The two transformations behave as in the following examples:

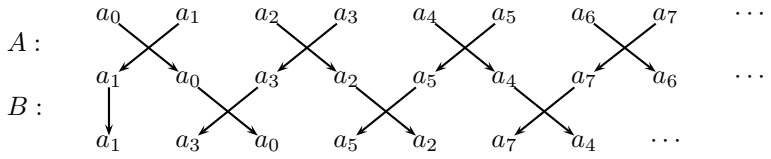
$$\mathscr{T}_A([a_0, a_1, a_2, a_3, a_4, a_5, a_6, \ldots]) = [a_1, a_0, a_3, a_2, a_5, a_4, a_7, \ldots]$$

$$\mathscr{T}_B([b_0, b_1, b_2, b_3, b_4, b_5, \ldots]) = [b_0, b_2, b_1, b_4, b_3, b_6, \ldots]$$

When we compose $\mathscr{T}_A$ and $\mathscr{T}_B$ we get the following transformation:

$$\mathscr{T}_{A\circ B}([a_0, a_1, a_2, a_3, a_4, a_5, a_6, \ldots]) = [a_1, a_3, a_0, a_5, a_2, a_7, \ldots]$$

Intuitively, looking at $\mathscr{T}_{A\circ B}$ we can see that no finite lookahead seems to suffice for this function. The following argument is illustrated by this figure:



Formally, for each a_i such that $i \geq 0$, $\mathscr{T}_{A\circ B}$ is the following function:

- if $i = 1$, a_i is output at position 0;
- if i is even and greater than 1, a_i is output at position $i - 2$;
- if i is equal to $k-2$ where k is the length of the input, a_i is output at position $k - 1$;
- if i is odd and different from $k - 2$, a_i is output at position $i + 2$.

It is easy to see that the above transformation cannot be computed by any ESFT. Let's assume by contradiction that there exists an ESFT that computes $\mathscr{T}_{A\circ B}$. We consider the ESFT C with minimal lookahead (let's say n) that computes $\mathscr{T}_{A\circ B}$.

We now show that on an input of length greater than $n+2$, C will misbehave. The first transition of C that will apply to the input will have a lookahead of size $l \leq n$. We now have three possibilities (the case $n = k - 2$ does not apply due to the length of the input):

$l = 1$**:** before outputting a_0 (at position 2) we need to output a_1 and a_3 which we have not read yet. Contradiction;

l **is odd:** position $l + 1$ is receiving a_{l-1} therefore C must output also the elements at position l. Position l should receive a_{l+2} which is not reachable with a lookahead of just l. Contradiction;

l **is even and greater than** 1**:** since $l > 1$, position l is receiving a_{l-2}. This means C is also outputting position $l - 1$. Position $l - 1$ should receive a_{l+1} which is not reachable with a lookahead of just l. Contradiction;

We now have that n cannot be the minimal lookahead which contradicts our initial hypothesis. Therefore $\mathscr{T}_{A\circ B}$ is not ESFT-definable. ⊠

2.2 Symbolic Transducers with Registers

Registers provide a practical generalization of SFTs. SFTs with registers are called STs, since their state space (reachable by registers) may no longer be finite. An ST uses a *register* as a symbolic representation of states in addition to explicit (control) states. The rules of an ST are guarded commands with a symbolic input and output component that may use the register. By using Cartesian product types, multiple registers are represented with a single (compound) register. Equivalence of STs is undecidable but STs are closed under composition [20].

Definition 7. A *Symbolic Transducer* or *ST* over $\sigma \to \gamma$ and register type τ is a tuple $A = (Q, q^0, \rho^0, R)$,

- Q is a finite set of *states*;
- $q^0 \in Q$ is the *initial state*;
- $\rho^0 \in \mathcal{U}^\tau$ is the *initial register value*;
- R is a finite set of *rules* $R = \Delta \cup F$;
- Δ is a set of *transitions* $r = (p, \varphi, o, u, q)$, also denoted $p \xrightarrow{\varphi/o;u} q$,
  - $p \in Q$ is the *start* state of r;
  - φ, the *guard* of r, is a $(\sigma \times \tau)$-predicate;
  - o, the *output* of r, is a finite sequence of $((\sigma \times \tau) \to \gamma)$-terms;
  - u, the *update* of r, is a $((\sigma \times \tau) \to \tau)$-term;
  - $q \in Q$ is the *end* state of r.
- F is a set of *final rules* $r = (p, \varphi, o)$, also denoted $p \xrightarrow{\varphi/o} \bullet$,
  - $p \in Q$ is the *start* state of r;
  - φ, the *guard* of r, is a τ-predicate;
  - o, the *output* of r, is a finite sequence of $(\tau \to \gamma)$-terms.

All ST rules in R have lookahead 1 and all final rules have lookahead 0. Longer lookaheads are not needed because registers can be used to record history, in particular they may be used to record previous input characters. A canonical way to do so is to let τ be σ^* that records previously seen characters, where initially $\rho^0 = []$, indicating that no input characters have been seen yet.

An ESFT transition

$$p \xrightarrow[3]{\lambda(x_0,x_1,x_2).\varphi(x_0,x_1,x_2)/\lambda(x_0,x_1,x_2).o(x_0,x_1,x_2)} q$$

can be encoded as the following set of ST rules where p_1 and p_2 are new states

$$p \xrightarrow{(\lambda(x,y).\mathrm{t})/[];\lambda(x,y).cons(x,[])} p_1 \quad p_1 \xrightarrow{(\lambda(x,y).\mathrm{t})/[];\lambda(x,y).cons(x,y)} p_2$$
$$p_2 \xrightarrow{(\lambda(x,y).\varphi(y[1],y[0],x))/\lambda(x,y).o(y[1],y[0],x);\lambda(x,y).[]} q$$

Final rules are encoded similarly. The only difference is that q above is $\bullet$ and the register updated is not used in the third rule. An ST rule $(p, \varphi, o, u, q) \in R$ denotes the following set of concrete transitions:

$$[\![(p, \varphi, o, u, q)]\!] \stackrel{\text{def}}{=} \{(p, s) \xrightarrow{a/[\![o]\!](a,s)} (q, [\![u]\!](a, s)) \mid (a, s) \in [\![\varphi]\!]\}$$

A final ST rule $(p, \varphi, o) \in F$ denotes the following set of concrete transitions:

$$[\![(p, \varphi, o)]\!] \stackrel{\text{def}}{=} \{(p, s) \xrightarrow{[]/[\![o]\!](s)} \bullet \mid s \in [\![\varphi]\!]\}$$

The reachability relation $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\!\rightarrow_A \; q$ for $\boldsymbol{a} \in \Sigma^*$, $\boldsymbol{b} \in \Gamma^*$, $p \in (Q \times \mathcal{U}^\tau)$, $q \in (Q \times \mathcal{U}^\tau) \cup \{\bullet\}$ is defined analogously to ESFTs and $\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \mid (q^0, \rho^0) \xrightarrow{\boldsymbol{a}/\boldsymbol{b}} \bullet\}$.

3 Register Elimination

The main advantage of STs is their succinctness and the fact that Bek programs can directly be mapped to STs. The downside of using STs is that many of the desired properties, such as equivalence, idempotence, and commutativity, are no longer decidable. One approach to decide those properties is to transform STs to SFTs by exploring all the possible register values. However, this is only possible for finite alphabets and in general not feasible due to state space explosion. This is where ESFTs play a central role.

In this section we describe an algorithm that allows us to transform a class of STs into ESFTs. The algorithm has several applications.

One application is to eliminate registers at the expense of increasing the lookahead. The algorithm can be applied to a class of *product* STs with *two outputs*. The algorithm is agnostic about how the output looks like. Product STs are used in the single-valuedness checking algorithm of SFTs.

While ESFTs provide a powerful generalization of SFTs, they are unfortunately not closed under composition as shown in Section 2.1. Another application of the transformation algorithm is a semi-decision procedure for composing ESFTs. The technique is to first translate the ESFTs to STs, as outlined in Section 2.2, then compose the STs, and finally apply the register elimination algorithm to convert the composed ST back to an ESFT, if possible. We are currently investigating which subclasses of ESFT are effectively closed under composition.

The core idea that underlies the register elimination algorithm is a symbolic generalization of the classical state elimination algorithm for converting an NFA to a regular expression (see e.g. [22, Section 3.3]), that uses the notion of extended automata whose transitions are labelled by regular expressions. Here the labels of the ST are predicates over sequences of elements of fixed lookahead. Essentially the intermediate data structure of the algorithm is an Extended ST.

Input: ST $A^{\sigma/\gamma;\tau}$.
Output: $\perp$ or an ESFT over $\sigma \to \gamma$ that is equivalent to A.

1. Lift A to use the input sort σ^* by replacing each transition $p \xrightarrow{\varphi/o;u} q$ with the following transition annotated with a lookahead of 1 and with $x : \sigma^*$,

$$p \xrightarrow[1]{\lambda(x,y).x \neq [] \wedge \varphi(x[0],y)/\lambda(x,y).o(x[0],y);\lambda(x,y).u(x[0],y)} q$$

(Apply similar transformation to final rules and give them lookahead 0.)

2. Repeat the steps 2.a-2.c while there exists a state that does not have a self loop (a self loop is a transition whose start and end states are equal).

2.a Choose a state p, such that p is not the state of any self loop and p is not the initial state.

2.b Do for all transitions $p_1 \xrightarrow[k]{\varphi_1/o_1;u_1} p \xrightarrow[\ell]{\varphi_2/o_2:u_2} p_2$ in R:

let $\varphi = \lambda(x,y).\varphi_1(x,y) \wedge \varphi_2(tail^k(x), u_1(x,y))$
let $o = \lambda(x,y).o_1(x,y) \cdot o_2(tail^k(x), u_1(x,y))$
let $u = \lambda(x,y).u_2(tail^k(x), u_1(x,y))$
if $IsSat(\varphi)$ **then let** $r = p_1 \xrightarrow[k+\ell]{\varphi/o;u} p_2$ and add r as a new rule

2.c Delete the state p.

3. If no guard and no output depends on the register, remove the register from all the rules in the ST and return the resulting ST as an ESFT, otherwise return $\perp$.

After the first step, the original ST accepts an input $[a_0, a_1, a_2]$ and produces output v iff the transformed ST accepts $[cons(a_0, \_), cons(a_1, \_), cons(a_2, \_)]$ and produces output v, where the tails $\_$ are unconstrained and irrelevant. Step 2 further groups the inputs characters, e.g., to $[cons(a_0, cons(a_1, \_)), cons(a_2, \_)]$, etc, while maintaining this input/output property with respect to the original ST. Finally, in step 3, turning the ST into an ESFT, leads to elimination of the register as well as lowering of the character sort back to σ, and replacing each occurrence of $tail^k(x)$ with corresponding individual tuple element variable x_k. Soundness of the algorithm follows.

The algorithm omits several implementation aspects that have considerable effect on performance. One important choice is the order in which states are removed. In our implementation the states with lowest total number of incoming and outgoing rules are eliminated first. It is also important to perform the choices in an order that avoids unreachable state spaces. For example, the elimination of a state p in step 2 may imply that φ is unsatisfiable and consequently that p_2 is unreachable if the transition from p is the only transition leading to p_2. In this case, if p is reachable from the initial state, choosing p_2 before p in step 2 would be wasteful.

4 Unicode Case Study

In this section we show how to describe realistic encoding and decoding routines using STs. We use Bek as the concrete programming language for STs.

A *hextet* is a non-negative integer $< 2^{16}$, and an *octet* is a non-negative integer $< 2^8$, i.e., hextets correspond to 16-bit bitvectors and octets correspond to bytes. Hextets are used in modern programming and scripting languages to represent character codes. For example, in C# as well as in JavaScript, string representation involves arrays of characters, where each character has a unique numeric code in form of a hextet. For example, the JavaScript expression `String.fromCharCode(0x48, 0x65, 0x6C, 0x6C, 0x153, 0x21)` equals to the string "`Hellœ!`".

A Unicode *code point* is an integer between 0 and 1,112,064 ($\texttt{10FFFF}_{16}$). *Surrogates* are code points between $\texttt{D800}_{16}$ and $\texttt{DFFF}_{16}$ and are not valid character code points according to the UTF-8 definition.[2]

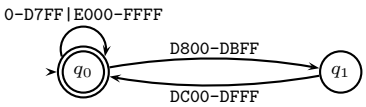
```
program utf8encode(input){
  return iter(c in input)[H:=false; r:=0;]
  {
    case (!H&&(0<=c)&&(c<=0x7F)): yield(c);                          //one octet
    case (!H&&(0x7F<c)&&(c<=0x7FF)):
      yield(0xC0|((c>>6)&0x1F), 0x80|(c&0x3F));                      //two octets
    case (!H&&(0x7FF<c)&&(c<=0xFFFF)&&((c<0xD800)||(c>0xDFFF))):
      yield(0xE0|((c>>12)&0xF), 0x80|((c>>6)&0x3F), 0x80|(c&0x3F)); //three octets
    case (H&&(0xDC00<=c)&&(c<=0xDFFF)):  H:=false; r:=0;             //low surrogate
      yield((0x80|(r << 4))|((c>>6)&0xF), 0x80|(c&0x3F));
    case (!H&&(0xD800<=c)&&(c<=0xDBFF)): H:=true; r:=c&3;            //high surrogate
      yield (0xF0|(((1+((c>>6)&0xF))>>2)&7),(0x80|(((1+((c>>6)&0xF))&3)<<4))|((c>>2)&0xF));
    case (true):  raise InvalidInput;
    end case (H): raise InvalidInput;
  };
}

program utf8decode(input){
  return iter(c in input)[q:=0; r:=0;]
  {
    case ((q==0)&&(0<=c)&&(c<=0x7F)):    yield (c);
    case ((q==0)&&(0xC2<=c)&&(c<=0xDF)): q:=3; r:=(c&0x3F)<<6;
    case ((q==0)&&(c==0xE0)):            q:=7;
    case ((q==0)&&(c==0xED)):            q:=6;
    case ((q==0)&&(0xE1<=c)&&(c<=0xEF)): q:=2; r:=(c&0xF)<<12;
    case ((q==0)&&(0xF1<=c)&&(c<=0xF3)): q:=1; r:=(c&7)<<8;
    case ((q==0)&&(c==0xF0)):            q:=4;
    case ((q==0)&&(c==0xF4)):            q:=5; r:=0x400;
    case ((q==1)&&(0x80<=c)&&(c<=0xBF)): q:=8; r:=0xD800|(((r|((c&0x30)<<2))-0x40)|((c&0x0F)<<2));
    case ((q==4)&&(0x90<=c)&&(c<=0xBF)): q:=8; r:=0xD800|((((c&0x30)<<2)-0x40)|((c&0x0F)<<2));
    case ((q==5)&&(0x80<=c)&&(c<=0x8F)): q:=8; r:=0xD800|(((r|((c&0x30)<<2))-0x40)|((c&0x0F)<<2));
    case ((q==2)&&(0x80<=c)&&(c<=0xBF)): q:=3; r:=r|((c&0x3F)<<6);
    case ((q==6)&&(0x80<=c)&&(c<=0x9F)): q:=3; r:=0xD000|((c&0x3F)<<6);
    case ((q==7)&&(0xA0<=c)&&(c<=0xBF)): q:=3; r:=(c&0x3F)<<6;
    case ((q==8)&&(0x80<=c)&&(c<=0xBF)): q:=3; yield(r|((c>>4)&3)); r:=0xDC00|((c&0xF)<<6);
    case ((q==3)&&(0x80<=c)&&(c<=0xBF)): q:=0; yield(r|(c&0x3F)); r:=0;
    case (true):        raise InvalidInput;
    end case (!(q==0)): raise InvalidInput;
  };
}
```

Fig. 1. UTF-8 encoder and decoder in Bek

UTF-16 is the standard character encoding used in modern programming and scripting languages. With UTF-16 format, Unicode symbols are represented either directly by hextets, or as pairs of hextets, so called *surrogate pairs*, that represent symbols in the upper Unicode range, e.g., the musical symbol 𝄵 called "cut time" has Unicode code point $\texttt{1D135}_{16}$ that is encoded in UTF-16 by the surrogate pair $[\texttt{D834}_{16}, \texttt{DD35}_{16}]$.

Not all sequences of hextets represent well-formed UTF-16 strings. Well-formed UTF-16 strings are precisely all those sequences of hextets that match the regular expression and corresponding symbolic finite automaton in Figure 2.

[2] `http://tools.ietf.org/html/rfc3629`




```
^([\0-\uD7FF\uE000-\uFFFF]|([\uD800-\uDBFF][\uDC00-\uDFFF]))*$
```

Fig. 2. UTF-16 validator. All numbers use hexadecimal notation and the range expression m-n is short for the predicate $\lambda x.m \leq x \leq n$.

Elements in the ranges `[\uD800-\uDBFF]` and `[\uDC00-\uDFFF]` are called *high surrogates* and *low surrogates*, respectively. A surrogate pair $[high, low]$ represents the Unicode symbol whose code point is $((high_{\langle 9,0\rangle} \ll 10)|low_{\langle 9,0\rangle}) + \texttt{10000}_{16}$, where $x_{\langle m,n\rangle}$ extracts bits m through n from x.

UTF-8 UTF-8 uses sequences of one up to four octets to encode single Unicode code points. Let c be a Unicode codepoint, the UTF-8 encoding of c is:

$$Utf8(c) \stackrel{\text{def}}{=} \begin{cases} [c], & \text{if } 0 \leq c \leq \texttt{7F}_{16}; \\ [\texttt{C0}_{16}|c_{\langle 10,6\rangle}, \texttt{80}_{16}|c_{\langle 5,0\rangle}], & \text{if } c \leq \texttt{7FF}_{16}; \\ [\texttt{E0}_{16}|c_{\langle 15,12\rangle}, \texttt{80}_{16}|c_{\langle 11,6\rangle}, \texttt{80}_{16}|c_{\langle 5,0\rangle}], & \text{if } c \leq \texttt{FFFF}_{16}; \\ [\texttt{F0}_{16}|c_{\langle 20,18\rangle}, \texttt{80}_{16}|c_{\langle 17,12\rangle}, \texttt{80}_{16}|c_{\langle 11,6\rangle}, \texttt{80}_{16}|c_{\langle 5,0\rangle}], & \text{otherwise.} \end{cases}$$

Some codepoints are not valid. In particular, in the third case, c may not be a surrogate, i.e., $c < \texttt{D800}_{16}$ or $c > \texttt{DFFF}_{16}$. Also, any number greater than $\texttt{10FFFF}_{16}$ is invalid. The exact details of how the UTF-8 encoding is computed from the UTF-16 encoding follows from the Bek program in Figure 1 and is discussed below.

Conversions. A UTF-16 to UTF-8 encoder takes a well-formed UTF-16 encoded string and converts it into the equivalent UTF-8 representation. Figure 1 shows the Bek program of such an encoder. The program makes essential use of bitwise operations over hextets, in particular, it uses bit shifting operations and logical bit operations. The input to the program is a sequence of hextets. The output produced by the encoder is a sequence of octets.

The Bek program represents a symbolic transducer that uses two registers: the Boolean register `H` and the character register `r`. The value of `H` is true if the previous character was a high surrogate, in which case the register `r` contains the two least significant bits of that high surrogate. When a low surrogate is input, it is used together with the value of `r` to yield the remaining two octets of the combined code point (the first two octets were output when the high surrogate was read). Both registers can be effectively eliminated by using an exploration algorithm. The resulting SFT is illustrated in Figure 3, where *HS* is the predicate for high surrogates and *LS* is the predicate for low surrogates. The rules of the SFT correspond to the different branches of the Bek program, where the exception cases correspond to the cases that are *disabled* at the respective states. What is quite remarkable is that the SFT has 11 rules and 5 states

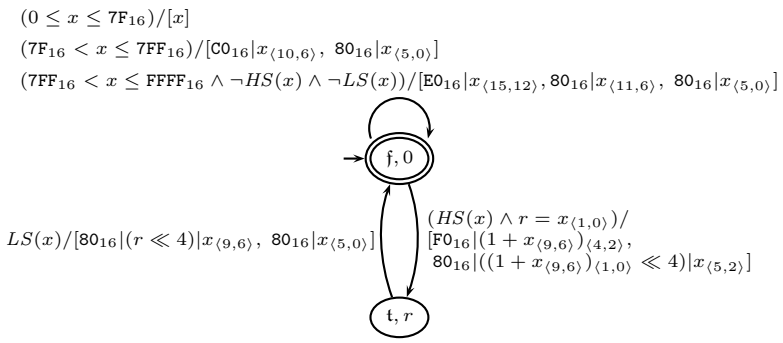


Fig. 3. SFT that is equivalent to the Bek program `utf8encode` in Figure 1. States are labelled by values of $(\texttt{H}, \texttt{r})$, there are five states: $(\mathfrak{f}, 0)$, $(\mathfrak{t}, 0)$, $(\mathfrak{t}, 1)$, $(\mathfrak{t}, 2)$, and $(\mathfrak{t}, 3)$.

in total, compared to an equivalent classical finite state transducer that would require 2^{16} transitions (one transition per hextet).

The program `utf8decode` in Figure 1 provides the inverse conversion from valid UTF-8 encoded sequences to valid UTF-16 encoded sequences. The equivalent SFT has in this case 1284 states and 6371 rules, that is in sharp contrast to the 5 states and 11 rules of the encoder.

5 Experiments and Evaluation

We first verify the functional correctness of *UTF8* encoder and decoder. Secondly, we analyze how the register elimination algorithm of Section 3 scales for different program's sizes.

5.1 Functional Correctness of Encoders and Decoders

The Bek programs in Figure 1 can be analyzed for various properties of interest by first converting them to STs and then to ESFT. While this analysis is very efficient for the ESFT of the encoder in Figure 3 it is more demanding for the decoder because of the size of the state space. As a fundamental correctness criterion, the valid input sequences of *utf8decode* should be the set of all valid UTF-8 sequences: $\mathscr{D}(\textit{utf8decode}) = \mathscr{D}(\textit{UTF8})$ where *UTF8* is the UTF-8 validator expressed as a Symbolic Automaton. An inspection of *UTF8* shows that the following validity properties are checked:

1. Octets $\texttt{C0}_{16}$, $\texttt{C1}_{16}$, $\texttt{F5}_{16}, \ldots, \texttt{FF}_{16}$ are disallowed.
2. Invalid combinations of start-octets and continuation-octets are disallowed.
3. Sequences that decode to a value with a shorter encoding (so called "overlong" sequences) are disallowed.
4. Sequences starting with $\texttt{F4}_{16}$ encoding a value $> \texttt{10FFFF}_{16}$ are disallowed.
5. Encodings of surrogates (having start-octet $\texttt{ED}_{16}$) are disallowed.

In particular, overlong encodings, such as the encoding $[\texttt{C0}_{16}, \texttt{AE}_{16}]$ of '.', are disallowed.

Moreover, we expect the decoder to perform the inverse of the encoder and vice versa. Let I be the identity SFT, i.e., I has a single state q_0 and a single rule $q_0 \xrightarrow{\lambda x.\mathrm{t}/[\lambda x.x]} q_0$. Let $E = \mathscr{T}_{utf8encode}$, $D = \mathscr{T}_{utf8decode}$, $U_{\mathrm{utf16}} = \mathscr{D}(UTF16)$, and $U_{\mathrm{utf8}} = \mathscr{D}(UTF8)$. The following must hold:

- $E \circ D \stackrel{1}{=} I$, $\mathscr{D}(E) = \mathscr{D}(E \circ D) = U_{\mathrm{utf16}}$
- $D \circ E \stackrel{1}{=} I$, $\mathscr{D}(D) = \mathscr{D}(D \circ E) = U_{\mathrm{utf8}}$

$A \stackrel{1}{=} B$, iff A and B produce the same output on each of the inputs in their domain intersection (1-equality [20]). Consider $D : U_{\mathrm{utf16}} \to U_{\mathrm{utf8}}$ and $E : U_{\mathrm{utf8}} \to U_{\mathrm{utf16}}$ as functions. Thus D and E are *bijections* and inverses of each other.

5.2 Use of Register Elimination

In general, an ESFT to SFT conversion is needed for deciding 1-equality. The previous technique eliminated the registers by fully exploring their reachable state space and created an SFT prior to invoking the equivalence algorithm [21]. The technique introduced here takes a step further. It gradually increases the lookahead of a *2-output ST* by shortcutting intermediate states in an attempt to completely eliminate register dependencies from the guards and the output terms. First, we convert ESFTs to STs by introducing registers, as explained in Section 2.2. Second, we compute a 2-output ST as a product of the two STs that has synchronized input and where infeasible guard combinations have been eliminated, corresponding to product-SFTs in [20, Definition 7]. Third, we compute an equivalent 2-output ESFT (when possible) from the 2-output ST using the register elimination algorithm explained in Section 3. Fourth, we convert that 2-output ESFT into a 2-output SFT whose characters are grouped into sequences of characters of given lookahead length and note that this transformation preserves one-equality of the original ESFTs due to the synchronized inputs. Finally, we apply a variation of the one-equality algorithm for SFTs to the 2-output product SFT that is the value of C in the one-equality algorithm [20, Figure 3]. The procedure described above might not terminate. However this has not been the case in our case study. We are currently investigating decidability of one-equality of ESFTs, and a more direct approach.

We illustrate the register elimination algorithm, in the case of composition, using some rules of the case study. Consider the composition $ED = E \circ D$ (encoding followed by decoding). The resulting ST ED uses a register (inherited from D), and has 5 states and 22 rules. Besides the initial state q_0, all other states q are non-final and *intermediate* in the following sense: all paths through q have the form:

$$q_0 \xrightarrow[1]{\varphi(x_0)/([],f(x_0))} q \xrightarrow[1]{\psi(x_0,y)/(\beta(x_0,y),0)} q_0$$

where the first rule is independent of the register (depends only on the input element x_0). The path can be represented with an equivalent rule with a lookahead of 2 elements:

$$q_0 \xrightarrow[2]{\varphi(x_0)\wedge\psi(x_1,f(x_0))/(\beta(x_1,f(x_0)),0)} q_0$$

After the removal of all such intermediate states from ED, the register y can be deleted from all the new rules because no guard or output depends on the register.[3]

Table 1 shows the running times of the operations needed to perform the checks described above.

Table 1. Running Times for Functional Correctness

Operation	Running Time
$\mathscr{D}(E) = U_{\text{utf16}}$	47 ms
$\mathscr{D}(E \circ D) = U_{\text{utf16}}$	109 ms
$\mathscr{D}(D) = U_{\text{utf8}}$	156 ms
$\mathscr{D}(D \circ E) = U_{\text{utf8}}$	320 ms
$E \circ D \overset{1}{=} I$ (exploration)	82,000 ms
$D \circ E \overset{1}{=} I$ (exploration)	134,000 ms
$E \circ D \overset{1}{=} I$ (reg. elim.)	**123** ms
$D \circ E \overset{1}{=} I$ (reg. elim.)	**215** ms

The first four entries of Table 1 show the running time for the domain checks. The times are all under 0.4 seconds. To perform the domain checks we compute the ESFAs (ESFTs with empty outputs) corresponding to domain of the ESFTs, and we check for automata equivalence. In order to improve the efficiency, when possible, we transform ESFAs into equivalent SFAs (SFTs with empty outputs) and use the equivalence algorithm for SFAs [20]. Without this transformation, some of the running times would be above 1 minute.

The next two entries of Table 1 show the running times for 1-equality with the exploration algorithm in [21]. It is clear from the data (> 100 seconds) that the state explosion causes the algorithm to work only on programs of small sizes. Finally, the last two entries of the table represent the running time for the improved algorithm of Section 3 with register elimination. It is important to point out that in this case we do not check for domain equivalence but only for 1-equality. Performing the check after the register elimination algorithm achieves a 600x speed-up against the full exploration version in this particular case study. This is a consequence of the succinctness of ESFTs.

[3] The online Bek tutorial `http://www.rise4fun.com/Bek/tutorial/utf8` contains further analysis scenarios.

5.3 Running Time Analysis with Register Elimination

In this section we run our register elimination algorithm on bigger program instances. Most of the checks performed in this section will time out (longer than 1 hour) when using the full exploration algorithm.

We consider consecutive compositions of encoders and decoders and analyze their correctness using 1-equality for ESFTs. This experiment is motivated by a common form of attack called *double encoding* [15]. This attack technique consists of encoding the user input twice in order to cause unexpected behaviour. We define the following notation for consecutive composition of STs. Given an ST P we define $P^1 \equiv P$ and $P^{i+1} \equiv P \circ P^i$. We verify the following properties and analyze their execution times.

Equivalence for Enc/Dec: $E^i \circ D^i \overset{1}{=} I$ for $1 \leq i \leq 9$, Figure 4(a);
Inequivalence for Enc/Dec: $E^{i+1} \circ D^i \overset{1}{\neq} I$ for $1 \leq i \leq 9$, Figure 4(a);
Equivalence for Dec/Enc: $D^i \circ E^i \overset{1}{=} I$ for $1 \leq i \leq 3$, Figure 4(b);
Inequivalence for Dec/Enc: $D^i \circ E^{i+1} \overset{1}{\neq} I$ for $1 \leq i \leq 3$, Figure 4(b).

Figure 4(a) shows the running times for the case in which we first encode and then decode. The figure plots the following measures where i varies between 1 and 9:

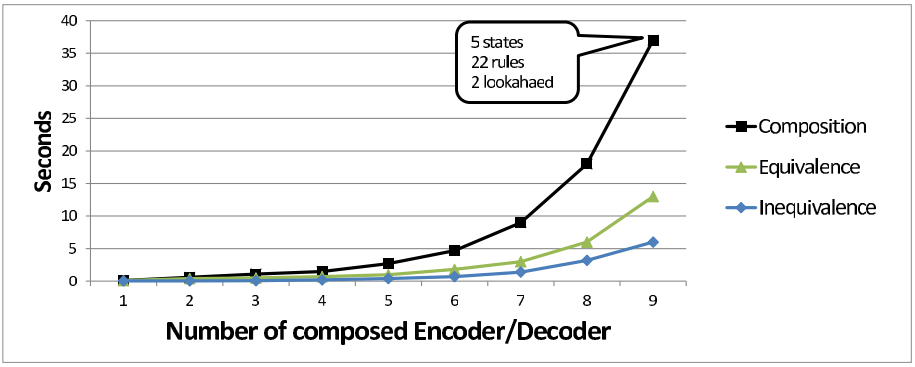
Composition: cost of computing $E^{i+1} \circ D^i$ (we omit the cost of computing $E^i \circ D^i$ since it is almost equivalent);
Equivalence: cost of checking $E^i \circ D^i \overset{1}{=} I$;
Inequivalence: cost of checking $E^{i+1} \circ D^i \overset{1}{\neq} I$.

In this case the algorithm scales pretty well with the number of STs. It is worth noticing that at every i we are analyzing the composition of $2i$ transducers in the case of equivalence and $2i + 1$ transducers in the case of inequivalence.

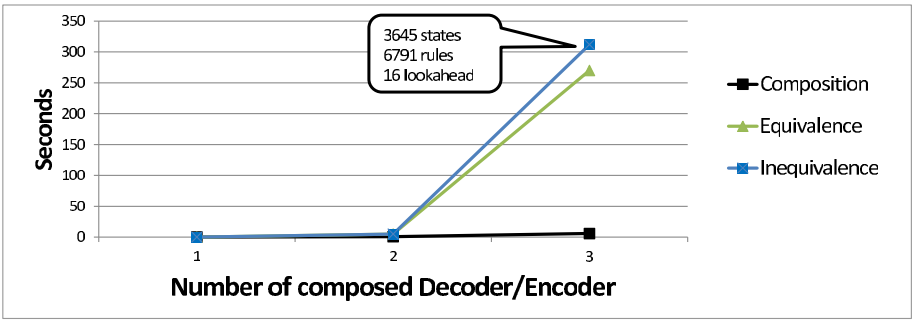
Figure 4(b) shows the running times for the case in which we first decode and then encode. The plot has the same meaning as before, but in this case the running time increases at a faster pace. This happens for two reasons: 1) the state space is bigger, and 2) the lookahead is bigger.

In the case in which we first encode the number of states and transitions does not grow when i increases. However, when we first decode, we early ($i = 3$) reach a big number of states (3645) and transitions (6791). Moreover, while the size of the lookahead in the first case remains the same (it is always 2), it grows exponentially with i when we first decode. Indeed for $i = 1, 2, 3$ we have lookaheads of size $4, 8, 16$ respectively. This causes the register elimination algorithm to explore more paths.

We noticed when we composed the encoder with the decoder, that the lookahead size and the number of states and transitions do not grow when i increases. However, Figure 4(a) shows that the running time grows exponentially in i. The complexity indeed does not only depend on the size of the ESFT, but also on the size of its predicates. The predicate sizes increase when we compose STs, causing the SMT solver to affect the performance of both the composition and equivalence algorithms, which perform several satisfiability checks on predicates.



(a) Encoding then Decoding



(b) Decoding then Encoding

Fig. 4. Running time in *seconds* for equivalence/inequivalence checking of multiple compositions of encoders and decoders

6 Related Work

Symbolic finite transducers (SFTs) and Bek were originally introduced in [9] with a focus on security analysis of sanitizers. The key properties that are studied in [9] from a practical point of view are idempotence, commutativity and equivalence checking of sanitizers. The formal foundations and the theoretical analysis of the underlying SFT algorithms, in particular, an algorithm for deciding equivalence of single-valued SFTs, modulo a decidable background theory is studied in [20], including a more general *1-equality* algorithm that factors out the decision problem for single-valuedness, and allows non-determinism without violating single-valuedness. The formalism of SFTs is extended in [20] to Symbolic Transducers (STs) that allow the use of registers. A "brute-force" exploration algorithm for register elimination is analyzed in [21]. However, the algorithm only copes with finite-ranged register updates and generally produces large state spaces. The focus and the motivation of the current paper is *efficient register elimination*. We introduce *extended symbolic finite transducers* (ESFTs)

which are strictly more expressive than SFTs. We then propose an algorithm that compiles a subclass of STs to ESFTs and that does not assume the input alphabet to be finite. Finally, the succinctness of ESFTs enables fast analysis of previously intractable (or not expressible) programs.

In recent years there has been considerable interest in automata using infinite alphabets [18], starting with the work on *register automata* [11]. Finite words over an infinite alphabet are often called *data words* in the literature. This line of work focuses on fundamental questions about definability, decidability, complexity, and expressiveness on classes of automata on one hand and fragments of logic on the other hand.

Streaming transducers [1] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence. Streaming transducers are largely orthogonal to SFTs or the extension of ESFTs, as presented in the current paper. For example, streaming transducers allow reversing the input, which is not possible with ESFTs, while arithmetic is not allowed in streaming transducers but plays a central role in our applications of ESFTs to string encoders.

We use the SMT solver Z3 [6] for incrementally solving label constraints that arise during the exploration algorithm. Similar applications of SMT techniques have been introduced in the context of symbolic execution of programs by using path conditions to represent under and over approximations of reachable states [8]. The distinguishing feature of our exploration algorithm is that it computes a precise transformation that is symbolic with respect to input labels, while allowing different levels of concretization with respect to the state variables. The resulting extended symbolic finite transducer is not an under or over approximation, but functionally *equivalent* to the original symbolic transducer. This is important for achieving a sound and complete analysis.

Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. Kaluza [17] extends the SMT solver to handle equations over strings and equations with multiple variables. We are not aware of previous work investigating the use of finite transducers for verifying code as complex as *utf8encoder* and *utf8decoder*. One obvious explanation for this is that classical finite transducers are not directly suited for this purpose; indeed, symbolic finite transducers can be exponentially more succinct than classical finite transducers with respect to alphabet size.

7 Conclusions

Several web applications assume the correctness of encoding and decoding functions. However, practical experience shows that writing correct encoders and decoders is a hard task. This paper presents an algorithmic extension of Bek, a language for writing, analyzing string manipulation routines. We introduce extended symbolic finite transducers (ESFTs) to enable analysis of previously intractable programs such as string decoders. We prove correctness of *UTF8* encoder and decoder, even in the case of double encoding. We show that our algorithms are fast in practice, and scale up to 20 encoder/decoder compositions.

References

1. Alur, R., Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: POPL 2011, pp. 599–610. ACM (2011)
2. Bek, http://research.microsoft.com/bek
3. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. In: Fontaine, P., Goel, A. (eds.) SMT 2012, pp. 76–86 (2012)
4. Bjørner, N., Tillmann, N., Voronkov, A.: Path Feasibility Analysis for String-Manipulating Programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
5. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
6. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Fülöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers. EATCS. Springer (1998)
8. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007, pp. 47–54 (2007)
9. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with Bek. In: Proceedings of the USENIX Security Symposium (August 2011)
10. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011)
11. Kaminski, M., Francez, N.: Finite-memory automata. TCS 134(2), 329–363 (1994)
12. Livshits, B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: PLDI 2009, pp. 75–86. ACM (2009)
13. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW 2005: Proceedings of the 14th International Conference on the World Wide Web, pp. 432–441 (2005)
14. NVD, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-2938
15. OWASP. Double encoding, https://www.owasp.org/index.php/Double_Encoding
16. SANS. Malware faq, http://www.sans.org/security-resources/malwarefaq/w-nt-unicode.php
17. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26 (March 2010)
18. Segoufin, L.: Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
19. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: ICST 2010, pp. 498–507. IEEE (2010)
20. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: Algorithms and applications. In: POPL 2012, pp. 137–150 (2012)
21. Veanes, M., Molnar, D., Mytkowicz, T., Livshits, B.: Data-parallel string-manipulating programs. Technical Report MSR-TR-2012-72, Microsoft Research (2012)
22. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, pp. 41–110. Springer (1997)

Robustness Analysis of Networked Systems*

Roopsha Samanta[1], Jyotirmoy V. Deshmukh[2], and Swarat Chaudhuri[3]

[1] University of Texas at Austin
roopsha@cs.utexas.edu
[2] University of Pennsylvania
djy@cis.upenn.edu
[3] Rice University
swarat@rice.edu

Abstract. Many software systems are naturally modeled as networks of interacting elements such as computing nodes, input devices, and output devices. In this paper, we present a notion of robustness for a networked system when the underlying network is prone to errors. We model such a system $\mathcal{N}$ as a set of processes that communicate with each other over a set of internal channels, and interact with the outside world through a fixed set of input and output channels. We focus on network errors that arise from channel perturbations, and assume that we are given a worst-case bound $\boldsymbol{\delta}$ on the number of errors that can occur in the internal channels of $\mathcal{N}$. We say that the system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-robust if the deviation of the output of the perturbed system from the output of the unperturbed system is bounded by $\boldsymbol{\epsilon}$.

We study a specific instance of this problem when each process is a Mealy machine, and the distance metric used to quantify the deviation from the desired output is either the L_1-norm or the Levenshtein distance (also known as the edit distance). For the former, we present a decision procedure for $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-robustness that is polynomial in the size of the network. For the latter, we present a decision procedure that is polynomial in the size of the network and exponential in the error bound on the output channel. Our solution draws upon techniques from automata theory, essentially reducing the problem of checking $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-robustness to the problem of checking emptiness for a certain class of reversal-bounded counter automata.

1 Introduction

More than ever before, we live in an era where computation does not exist in a vacuum, but is tightly integrated with networked communication and, often, interactions with the physical world. The heterogeneous systems that result from such integration — medical devices, power plants, vehicles and aircrafts — are often safety-critical. Unsurprisingly, they have long been regarded as important targets for formal methods.

* This research was partially supported by CCC-CRA Computing Innovation Fellows Project, NSF Award 1162076 and NSF CAREER award 1156059.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 229–247, 2013.

One aspect of such systems that has received relatively less attention in the formal methods literature is *uncertainty*. Uncertainty is pervasive in complex heterogeneous systems—for example, the data generated by sensors in the system can be inexact or corrupted, or the network channels implementing communication between the components of the system can be corrupt or lose data packets. Left unchecked, such uncertainty can wreak havoc. For a large class of systems, we often wish to determine to what extent the system behavior is predictable, when the system faces uncertainty. Traditional system correctness properties like safety and liveness are *qualitative* assertions about individual system traces, and proof techniques for such properties typically do not provide any *quantitative* measure on predictable system execution. In most engineering disciplines, the core property in reasoning about uncertain system behavior is *robustness*: "small perturbations to the operating environment or parameters of the system does not change the system's observable behavior substantially." This property is *differential*, in the sense that it relates a range of system traces possible under uncertainty. Furthermore, proof techniques to prove robustness demand a departure from traditional correctness checking algorithms as they require quantitative reasoning about the system behavior.

Given the above, formal reasoning about robustness of systems is a problem of practical as well as conceptual importance. In well-established areas such as control theory, robustness has always been a fundamental concern; in fact, there is an entire sub-area of control — *robust control* — that extensively studies this problem. However, as robust control typically involves reasoning about continuous state-spaces, the techniques and results therein are not directly applicable to cyber-physical systems which contain large amounts of discretized, discontinuous behavior.

In the context of cyber-physical systems, robustness analysis has only recently begun to gain attention. While several recent papers explore quantitative formal reasoning about robustness of software, the problem of reasoning about robustness with respect to errors in *networked communication* has been largely ignored. This is unfortunate as communication between different computation nodes is a fundamental feature of most modern systems. In particular, it is a key feature in emerging cyber-physical systems [20,21] where runtime error-correction features for ruling out uncertainty may not be an option. In this paper, we focus on such networked systems, and characterize an efficiently verifiable notion of robustness for them. At a high level, our contributions in this paper are as follows:

1. We present a model for communicating processes that is representative of the complexity of real systems.
2. We present a model for perturbations in the communication channels in the network, and formulate a notion of robustness for a networked system in the presence of these unreliable channels.
3. We present efficient, automata-theoretic, decision procedures for analyzing the robustness of the networked system with respect to different metrics characterizing the deviation of the observed behavior of the system.

In this paper, we model a synchronous, networked, system $\mathcal{N}$ as a set of communicating Mealy machines (processes). Processes communicate over a set of internal

channels, and interact with the outside world through a set of external input and output channels. We assume that processes communicate with each other using symbols from a finite alphabet, and perform computations over strings of such symbols. Each such symbol can be treated as an abstraction of complex data used for computation and communication in a real-world networked system.

As observed in [9], a critical requirement in networked cyber-physical systems is for all components of the system to have a common sense of time. This is usually ensured by protocols that guarantee that the global clock remains consistent across components. Thus, bearing this observation in mind, and following in the footsteps of recent papers on wireless control networks [20,21], classic models like Kahn process networks [16], and languages like Esterel [3], we assume our networks to have synchronous communication.

An input to the networked system $\mathcal{N}$ is a word capturing the sequence of symbols appearing on the input channels of $\mathcal{N}$; the externally observable behavior of $\mathcal{N}$ is a sequence of symbols appearing on its output channels. We model uncertainty by letting each internal channel perturb the data that is sent through it at any given point. Perturbations can include deletion of symbols and mutating symbols to other symbols. Deviations from the ideal, unperturbed, system's observable behavior are defined using suitable distance metrics on words. In this paper, we consider two such metrics: *Levenshtein distance* and the L_1*-norm*. We define the networked system $\mathcal{N}$ to be $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-*robust* if for any given input, the maximum change to the observable behavior of $\mathcal{N}$ is bounded by $\boldsymbol{\epsilon}$ as long as the number of perturbations introduced by the internal channels of $\mathcal{N}$ is bounded by $\boldsymbol{\delta}$.

Our central technical result is a decision procedure for determining whether a given system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-*robust*. Our algorithm reduces this problem to the problem of checking the emptiness of a certain class of *reversal-bounded counter automata*. A key step in our algorithm is the construction of automata that accept pairs of strings (s, t) if and only if the distance between s and t (w.r.t. a chosen metric) exceeds a specified constant. We present constructions for such automata for Levenshtein distance and the L_1-norm metric. We remark we can check robustness of a networked system with respect to any metric for which such automata constructions are possible. For the Levenshtein distance metric, the complexity of our algorithm is polynomial in the size of $\mathcal{N}$ and exponential in the error bound $\boldsymbol{\epsilon}$ on the output channels. For the L_1-norm distance metric, the complexity of our algorithm is polynomial in the size of $\mathcal{N}$.

The rest of the paper is organized as follows. In Sec. 2 we define our model of robust networked systems. In Sec. 3 and Sec. 4, we present the automata constructions involved in our decision procedure for checking robustness of such systems. We discuss related work in Sec. 5, and conclude with a discussion of future work in Sec. 6.

2 Robust Networked Systems

In this section, we present a formal model for a synchronous networked system. We then introduce a notion of robustness for computations of such networked

systems when the communication channels are prone to errors. In what follows, we use the following notation. Strings are typically denoted by lowercase letters s, t etc., with output strings sometimes denoted by primed lowercase letters s', t' etc. We denote the concatenation of strings s and t by $s.t$, the j^{th} character of string s by $s[j]$, the substring $s[i].s[i+1]\ldots.s[j]$ by $s[i,j]$, the length of the string s by $|s|$, and the empty string by λ. We sometimes denote vectors of objects using bold letters such as $\mathbf{s}$ and $\boldsymbol{\epsilon}$, with the j^{th} object in the vector denoted s_j and ϵ_j respectively.

2.1 Synchronous Networked System

A networked system, denoted $\mathcal{N}$, can be described as a directed graph $(\mathcal{P},\mathcal{C})$, with a set of processes $\mathcal{P} = \{P_1,\ldots,P_n\}$ and a set of communication channels $\mathcal{C}$. The set of channels consists of internal channels N, external input channels I, and external output channels O. An internal channel $C_{ij} \in N$ connects a source process P_i to a destination process P_j. An input channel has no source process, and an output channel has no destination process in $\mathcal{N}$.

Process Definition and Semantics. A process P_i in the networked system is defined as a tuple (In_i, Out_i, M_i), where $In_i \subseteq (I \cup N)$ is the set of P_i's input channels, $Out_i \subseteq (O \cup N)$ is the set of P_i's output channels, and M_i is a machine describing P_i's input/output behavior. We assume a synchronous model of computation: (1) at each tick of the system, each process consumes an input symbol from each of its input channels, and produces an output symbol on each its output channels, and (2) message delivery through the channels is instantaneous. We further assume that a networked system $\mathcal{N}$ has a computation alphabet Σ for describing the inputs and outputs of each process, and for describing communication over the channels. Please see Fig. 2.1 for an example networked system. Observe that a process may communicate with one, many or all processes in $\mathcal{N}$ using its output channels. Thus our network model subsumes unicast, multicast and broadcast communication schemes.

In this paper, we focus on processes described as Mealy machines. Recall that a Mealy machine [19] M is a deterministic finite-state transducer that in each step, reads an input symbol, possibly changes state, and generates an output symbol. Formally, M is described as a tuple $(\Sigma_{in}, \Sigma_{out}, Q, q_0, R)$, where Σ_{in} and Σ_{out} are input and output alphabets respectively, Q is a finite, nonempty set of states, q_0 is an initial state, and $R \subseteq Q \times \Sigma_{in} \times \Sigma_{out} \times Q$ is the transition function.

The operational semantics of M is defined in terms of its run $\rho(s)$ on an input string $s = s[1]\ldots s[m]$. A run is a sequence of the form (q_0,λ), $(q_1,s'[1])$, …, $(q_m,s'[m])$, where for each j, $1 \le j \le m$, $(q_{j-1}, s[j], s'[j], q'_j) \in R$. Such a run $\rho(s)$ of a Mealy machine defines the output function $[\![M]\!] : \Sigma^\star_{in} \to \Sigma^\star_{out}$, with $[\![M]\!](s[1].s[2]\ldots s[m]) = s'[1].s'[2].\ldots.s'[m]$.

In each tick, a Mealy machine process in a networked system $\mathcal{N}$ consumes a composite symbol (the tuple of symbols on its input channels), and outputs a composite symbol (the tuple of symbols on its output channels). Thus, the

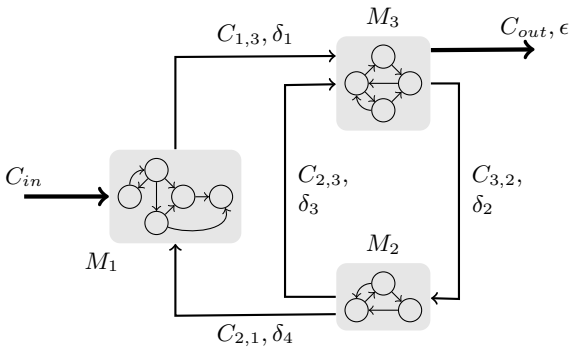


Fig. 2.1. Networked System

input alphabet Σ_{in} for M_i is $\Sigma^{|In_i|}$, and the output alphabet Σ_{out} is $\Sigma^{|Out_i|}$. Let $(\Sigma^{|In_i|}, \Sigma^{|Out_i|}, Q_i, q_{0_i}, R_i)$ be the tuple describing the Mealy machine underlying process P_i.

Operational Semantics of a Network. We define a *network state* $\mathbf{q}$ as the tuple $(q_1, \ldots, q_n, c_1, \ldots, c_{|N|})$, where for each i, $q_i \in Q_i$ is the state of P_i, and for each k, c_k is the state of the k^{th} internal channel, i.e., the current symbol in the channel. A transition of $\mathcal{N}$ has the following form:

$$(q_1, \ldots, q_n, c_1, \ldots, c_{|I|}) \xrightarrow{(a_1, \ldots, a_{|I|}), (a'_1, \ldots, a'_{|O|})} (q'_1, \ldots, q'_n, c'_1, \ldots, c'_{|I|})$$

Here $(a_1, \ldots, a_{|I|})$ denote the symbols on the external input channels, and $(a'_1, \ldots, a'_{|O|})$ denote the symbols on the external output channels. During a transition of $\mathcal{N}$, each process P_i consumes a composite symbol (given by the states of all internal channels in In_i and the symbols in the external input channels in In_i), changes state from q_i to q'_i, and outputs a composite symbol. The generation of an output symbol by P_i causes an update to the states of all internal channels in Out_i and results in the output of a symbol on each output channel in Out_i.

Thus, we can view the networked system $\mathcal{N}$ itself as a machine that in each step, consumes an $|I|$-dimensional input symbol $\mathbf{a}$ from its external input channels, changes state according to the transition functions R_i of each process, and outputs an $|O|$-dimensional output symbol $\mathbf{a}'$ on its external output channels.

Formally, we define the semantics of a computation of $\mathcal{N}$ using the tuple $(\Sigma^{|I|}, \Sigma^{|O|}, Q, \mathbf{q}_0, R)$, where $Q = (Q_1 \times \ldots \times Q_n \times \Sigma^{|N|})$ is the set of states and $R \subseteq (Q \times \Sigma^{|I|} \times \Sigma^{|O|} \times Q)$ is the network transition function. The initial state

$\mathbf{q}_o = (q_{0_1}, \ldots, q_{0_n}, c_{0_1}, \ldots, c_{0_{|N|}})$ of $\mathcal{N}$ is given by the initial process states and internal channel states. An execution $\rho(\mathbf{s})$ of $\mathcal{N}$ on an input string $\mathbf{s} = \mathbf{s}[1]\mathbf{s}[2]\ldots\mathbf{s}[m]$ is defined as a sequence of configurations of the form $(\mathbf{q}_0, \lambda)$, $(\mathbf{q}_1, \mathbf{s}'[1]), \ldots, (\mathbf{q}_m, \mathbf{s}'[m])$, where for each j, $1 \leq j \leq m$, $(\mathbf{q}_{j-1}, \mathbf{s}[j], \mathbf{s}'[j], \mathbf{q}_j) \in R$. The output function computed by the networked system $[\![\mathcal{N}]\!] : (\Sigma^{|I|})^\star \rightarrow (\Sigma^{|O|})^\star$ is then defined such that $[\![\mathcal{N}]\!](\mathbf{s}[1].\mathbf{s}[2]\ldots\mathbf{s}[m]) = \mathbf{s}'[1].\mathbf{s}'[2]\ldots\mathbf{s}'[m]$.

2.2 Channel Perturbations and Robustness

An execution of a networked system is said to be perturbed if one or more of the internal channels are perturbed one or more times during the execution. A channel perturbation can be modeled as a deletion or substitution of the current symbol in the channel. To model symbol deletions[1], we extend the alphabet of each internal channel to $\Sigma_\lambda = \Sigma \cup \lambda$. A perturbed execution includes transitions corresponding to channel perturbations, of the form:

$$\begin{array}{c}(q_1, \ldots, q_n, c_1, \ldots, c_{|I|}) \\ \Big\downarrow \boldsymbol{\lambda}, \boldsymbol{\lambda} \\ (q'_1, \ldots, q'_n, c'_1, \ldots, c'_{|I|}),\end{array}$$

Here, for each i, the states q'_i and q_i are identical, and for some k, $c_k \neq c'_k$. Such transitions, termed τ-transitions[2], do not consume any input symbol and model instantaneous channel errors. We say that the k^{th} internal channel is perturbed in a τ-transition if $c_k \neq c'_k$. A perturbed network execution $\rho_\tau(\mathbf{s})$ on an input string $\mathbf{s} = \mathbf{s}[1]\mathbf{s}[2]\ldots\mathbf{s}[m]$ is a sequence of configurations $(\mathbf{q}_0, \boldsymbol{\lambda}), \ldots, (\mathbf{q}_\tau, \mathbf{s}'[m])$, where for any j either $(\mathbf{q}_{j-1}, \mathbf{s}[\ell], \mathbf{s}'[\ell], \mathbf{q}_j) \in R$ or $(\mathbf{q}_{j-1}, \boldsymbol{\lambda}, \boldsymbol{\lambda}, \mathbf{q}_j)$ is a τ-transition.

Note that there can be several possible perturbed executions of $\mathcal{N}$ on a string $\mathbf{s}$ which differ in their exact instances of τ-transitions and the channels perturbed in each instance. Each such perturbed execution generates a different perturbed output. For a specific perturbed execution $\rho_\tau(\mathbf{s})$ of the form $(\mathbf{q}_0, \boldsymbol{\lambda}), (\mathbf{q}_1, \mathbf{s}'[1]), \ldots, (\mathbf{q}_\tau, \mathbf{s}'[m])$, we denote the string $\mathbf{s}' = \mathbf{s}'[1].\mathbf{s}'[2]\ldots\mathbf{s}'[m]$ output by $\mathcal{N}$ along that execution by $[\![\rho_\tau]\!](\mathbf{s})$. We denote by $[\![\mathcal{N}_\tau]\!](\mathbf{s})$ the *set* of all possible perturbed outputs corresponding to the input string $\mathbf{s}$. Formally, $[\![\mathcal{N}_\tau]\!](\mathbf{s})$ is the set $\{\mathbf{s}' \mid \exists \rho_\tau(\mathbf{s}) \text{ s.t. } \mathbf{s}' = [\![\rho_\tau]\!](\mathbf{s})\}$.

[1] Note that though a perturbation can cause a symbol on an internal channel to get deleted in a given step, we expect that the processes reading from this channel will output a nonempty symbol in that step. In this sense, we treat an empty input symbol simply as a special symbol, and assume that each process can handle such a symbol.

[2] Note that a network transition of the form $((q_1, \ldots, q_n, c_1, \ldots, c_{|N|}), \boldsymbol{\lambda}, \mathbf{a}', (q'_1, \ldots, q'_n, c'_1, \ldots, c'_{|N|}))$ where for some i, $q_i \neq q'_i$ is not considered a τ-transition: such a transition involves a state change by some process on an empty input symbol along with the generation of a nonempty output symbol.

Robustness. A distance metric $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ over a set Σ^* of strings is a function with the following properties: $\forall s, t, u \in \Sigma^*$: (1) $d(s,t) = 0$ iff $s = t$, (2) $d(s,t) = d(t,s)$, and (3) $d(s,u) \leq d(s,t) + d(t,u)$. Let d be such a distance metric over strings. We extend the metric to vectors of strings in the standard fashion. Let $\mathbf{w} = (w_1, \ldots, w_L)$ be a vector of strings; then $d(\mathbf{w}, \mathbf{v}) = (d(w_1, v_1), \ldots, d(w_L, v_L))$.

Let τ_k denote the number of perturbations in the k^{th} internal channel in $\rho_\tau(\mathbf{s})$. Then, the *channel-wise perturbation count* in $\rho_\tau(\mathbf{s})$, denoted $\|\rho_\tau(\mathbf{s})\|$ is given by the vector $(\tau_1, \ldots, \tau_{|N|})$. We define robustness of a networked system as follows.

Definition 2.1 (Robust networked system).
Given an upper bound $\boldsymbol{\delta} = \{\delta_1, \ldots, \delta_{|N|}\}$ on the number of possible perturbations in each internal channel, and an upper bound $\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_{|O|})$ on the acceptable error in each external output channel of a networked system $\mathcal{N}$, we say that $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-robust if:

$$\forall \mathbf{s} \in (\Sigma^{|I|})^\star, \forall \rho_\tau(\mathbf{s}) : \quad \|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta} \implies d(\ [\![\mathcal{N}]\!](\mathbf{s}), [\![\rho_\tau]\!](\mathbf{s})\ ) \leq \boldsymbol{\epsilon}$$

3 Distance Tracking Automata

The above formulation of the robustness problem is independent of the metric used to measure the distance between strings in the output channels. In this paper, we focus on distance metrics such as the Levenshtein distance and L_1-norm that are the most prevalent metrics used in practice to measure distances between strings. In Sec. 4, we show that the robustness problem with respect to each of these metrics is efficiently analyzable by reducing it to the problem of checking language emptiness of a suitably constructed *reversal-bounded counter machine*. But first, we briefly review reversal-bounded counter machines, as we use them extensively in the rest of the paper.

3.1 Review: Reversal-bounded Counter Machines [14,15]

A (one-way, nondeterministic) h-counter machine $\mathcal{A}$ is a (one-way, nondeterministic) finite automaton, augmented with h integer counters. Let G be a finite set of integer constants (including 0). In each step, $\mathcal{A}$ may read an input symbol, perform a test on the counter values, change state, and increment each counter by some constant $g \in G$. A test on a set of integer counters $Z = \{z_1, \ldots, z_h\}$ is a Boolean combination of tests of the form $z\theta g$, where $z \in Z$, $\theta \in \{\leq, \geq, =, <, >\}$ and $g \in G$. Let $\mathcal{T}_Z$ be the set of all such tests on counters in Z.

Formally, $\mathcal{A}$ is defined as a tuple $(\Sigma_{in}, X, x_0, Z, G, E, F)$ where Σ_{in}, X, x_o, F, are the input alphabet, set of states, initial state, and final states respectively. Z is a set of h integer counters, and $E \subseteq X \times (\Sigma_{in} \cup \lambda) \times \mathcal{T}_Z \times X \times G^{|Z|}$ is the transition relation. Each transition $(x, \sigma, t, x', g_1, \ldots, g_h)$ denotes a change of state from x to x' on symbol $\sigma \in \Sigma_{in} \cup \lambda$, with $t \in \mathcal{T}_Z$ being the enabling test on the counter values, and $g_k \in G$ being the amount by which the k^{th} counter is incremented.

A configuration of a one-way multi-counter machine is defined as the tuple $(x, \sigma, z_1, \ldots, z_h)$, where x is the state of the automaton, σ is a symbol of the input string being read by the automaton and $z_1, \ldots, z_h$ are the values of the counters. We define a move relation $\rightarrow_{\mathcal{A}}$ on the configurations: $(x, \sigma, z_1, \ldots, z_h) \rightarrow_{\mathcal{A}} (x', \sigma', z_1', \ldots, z_h')$ iff $(x, \sigma, t(z_1, \ldots, z_h), x', g_1, \ldots, g_h) \in E$, where, $t(z_1, \ldots, z_h)$ is *true*, $\forall k$: $z_k' = z_k + g_k$, and σ' is the next symbol in the input string being read. A path is a finite sequence of configurations $\mu_1 \ldots, \mu_m$ where for all $j : \mu_j \rightarrow_{\mathcal{A}} \mu_{j+1}$. A string $s \in \Sigma_{in}^{\star}$ is accepted by $\mathcal{A}$ if there exists a path from $(x_0, s_0, 0, \ldots 0)$ to $(x, s_j, z_1, \ldots, z_h)$ for some $x \in F$ and $j \leq |s|$. The set of strings (language) accepted by $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A})$.

In general, multi-counter machines do not possess good algorithmic properties as they can simulate actions of Turing machines (even with just 2 counters). In [14], the author presents a class of counter machines that with certain restrictions on the counters possess efficiently decidable properties. We now briefly review these machines.

A counter is said to be in the increasing mode between two successive configurations if the value of the counter increases, and in the decreasing mode if the value of the counter decreases. We say that a counter *changes mode* if for (three) successive configurations, it goes from the increasing mode to the decreasing mode or vice versa. We say that a counter is *r-reversal bounded* if the maximum number of times it changes mode along *any* path is r. We say that a one-way multi-counter machine $\mathcal{A}$ is r-reversal bounded if each of its counters is at most r-reversal bounded. We denote the class of h-counter, r-reversal-bounded machines by NCM(h, r).

Lemma 3.1. *[12] The nonemptiness problem for a NCM(h, r) $\mathcal{A}$ can be solved in time polynomial in the size of $\mathcal{A}$ (i.e., the number of states of $\mathcal{A}$ and the number of counters h).*

In Sec. 4, we show how we can algorithmically construct composite machines that can check robustness of networked systems. A key component of these constructions are machines that accept a pair of strings iff the two strings are more than ϵ distance apart according to the chosen metric. We now present the construction of a deterministic finite automaton (DFA) $\mathcal{D}_{Lev}^{\epsilon}$ that accepts a pair of strings iff their Levenshtein distance is greater than ϵ, followed by the construction of a reversal-bounded counter automaton $\mathcal{D}_{L_1}^{\epsilon}$ that accepts a pair of strings iff their L_1-norm is greater than ϵ. In what follows, we assume that for all $i > |s|$, $s_i = \texttt{\#}$, where $\texttt{\#}$ is a special end-of-string symbol not in Σ. Let $\Sigma^{\texttt{\#}} = \Sigma \cup \{\texttt{\#}\}$.

3.2 Automaton for Tracking Levenshtein Distance

Levenshtein distance. The Levenshtein distance $d_{Lev}(s, t)$ between strings s and t is the minimum number of symbol insertions, deletions and substitutions

required to transform one string into another. The Levenshtein distance, or edit distance, is also defined by the following recurrence relations, for $i, j \geq 1$, and $s[0] = t[0] = \lambda$:

$$\begin{array}{l} d_{Lev}(s[0], t[0]) = 0, \qquad d_{Lev}(s[0,i], t[0]) = i, \qquad d_{Lev}(s[0], t[0,j]) = j \\ d_{Lev}(s[0,i],\ t[0,j]) = \min(\ d_{Lev}\,(s[0,i\text{-}1],\ t[0,j\text{-}1]) + \texttt{diff}(s[i],\ t[j]), \\ \qquad\qquad\qquad\qquad\quad d_{Lev}\,(s[0,i\text{-}1],\ t[0,j]) + 1, \\ \qquad\qquad\qquad\qquad\quad d_{Lev}\,(s[0,i],\ t[0,j\text{-}1]) + 1) \end{array} \tag{1}$$

Here, $\texttt{diff}(a, b)$ is defined to be 0 if $a = b$ and 1 otherwise. The first three relations, that involve empty strings, are obvious. The edit distance between the nonempty prefixes, $s[0, i]$ and $t[0, j]$, is the minimum of three distances: (1) the distance corresponding to editing $s[0, i\text{-}1]$ into $t[0, j\text{-}1]$ and substituting $s[i]$ for $t[j]$ if they are different symbols, (2) the distance corresponding to editing $s[0, i\text{-}1]$ into $t[0, j]$ and deleting $s[i]$, and, (3) the distance corresponds to editing $s[0, i]$ into $t[0, j\text{-}1]$ and inserting $t[j]$.

In [11], the authors show that for a given integer k, a relation $R \subseteq \Sigma^\star \times \Sigma^\star$ is rational if and only if for every $(s, t) \in R$, $|s| - |t| < k$. It is known from [10], that a subset is rational iff it is the behavior of a finite automaton. Thus, it follows from the above results that there exists a DFA that accepts the set of pairs of strings that are within bounded edit distance from each other. However, these theorems do not provide a constructive procedure for such an automaton. In what follows, we present a novel construction for a DFA $\mathcal{D}^{\epsilon}_{Lev}$ that accepts a pair of strings (s, t) iff $d_{Lev}(s, t) > \epsilon$.

The standard algorithm for computing the Levenshtein distance $d_{Lev}(s, t)$ uses a dynamic programming-based approach that uses the above recurrence relations. This algorithm organizes the bottom-up computation of the Levenshtein distance with the help of a table **tab** of height $|s|$ and width $|t|$. The 0^{th} row and column of **tab** account for the base case of the recursion. The $\mathbf{tab}(i, j)$ entry stores the Levenshtein distance of the strings $s[0, i]$ and $t[0, j]$. In general, the entire table has to be populated in order to compute $d_{Lev}(s, t)$. However, when one is only interested in some bounded distance ϵ, then for every i, the algorithm only needs to compute values for the cells from $\mathbf{tab}(i, i - \epsilon)$ to $\mathbf{tab}(i, i + \epsilon)$ [13]. We call this region the ϵ-*diagonal* of **tab**, and use this observation to construct the finite-state automaton $\mathcal{D}^{\epsilon}_{Lev}$.

The DFA $\mathcal{D}^{\epsilon}_{Lev}$ is defined to run on a pair of strings (s, t), and accept iff $d_{Lev}(s, t) > \epsilon$. In each step, $\mathcal{D}^{\epsilon}_{Lev}$ reads a pair of input symbols and changes state to mimic the bottom-up edit distance computation by the dynamic programming algorithm. We illustrate the operation of $\mathcal{D}^{\epsilon}_{Lev}$ with an example.

Example Run. A run of $\mathcal{D}^{\epsilon}_{Lev}$ on the string pair (s, t) that checks if $d_{Lev}(s, t) > \epsilon$, for $\epsilon = 2$ is shown in Fig. 3.1. After reading the i^{th} input symbol pair, $\mathcal{D}^{\epsilon}_{Lev}$ uses its state to remember the last $\epsilon = 2$ symbols of s and t that it has read, and transitions to a state that contains the values of $\mathbf{tab}(i, i)$ and the cells within the ϵ-diagonal, above and to the left of $\mathbf{tab}(i, i)$.

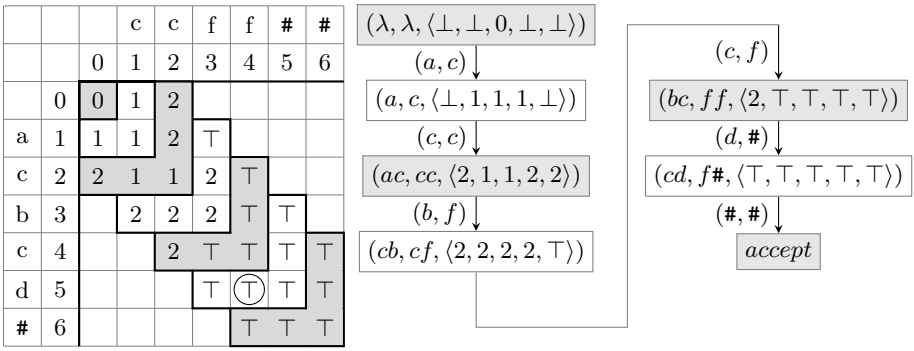


Fig. 3.1. Dynamic programming table emulated by $\mathcal{D}^{\epsilon}_{Lev}$. The table **tab** filled by the dynamic programming algorithm is shown to the left, and a computation of $\mathcal{D}^{\epsilon}_{Lev}$ on the strings $s = acbcd$ and $t = ccff$ is shown to the right. Here, $\epsilon = 2$.

Formally, $\mathcal{D}^{\epsilon}_{Lev}$ is defined as a tuple $(\Sigma^{\#} \times \Sigma^{\#}, Q_{Lev}, \mathbf{q}_{0\,Lev}, R_{Lev}, F_{Lev})$, where $(\Sigma^{\#} \times \Sigma^{\#})$, Q_{Lev}, $\mathbf{q}_{0\,Lev}$, R_{Lev}, F_{Lev} are the input alphabet, the set of states, the initial state, the transition relation and the set of final states respectively. $F_{Lev} = \{acc_{Lev}\}$ is a singleton set. In what follows, we define the other components.

We first note that as indicated earlier, $\mathcal{D}^{\epsilon}_{Lev}$ synchronously runs on a pair of strings, i.e., in each step it reads a symbol from $(\Sigma^{\#} \times \Sigma^{\#})$. We assume that each string is well-formed, i.e., each string is an element of $\Sigma^*.\#^*$. A state of $\mathcal{D}^{\epsilon}_{Lev}$ is defined as the tuple $(x, y, \mathbf{e})$, where x and y are strings of length at most ϵ and $\mathbf{e}$ is a vector containing $2\epsilon+1$ entries, with at most $\epsilon+3$ possible values for each entry. A state of $\mathcal{D}^{\epsilon}_{Lev}$ maintains the invariant that if i symbol pairs have been read, then $x = s[i\text{-}\epsilon\text{+}1, i]$, $y = t[i\text{-}\epsilon\text{+}1, i]$ and the entries in $\mathbf{e}$ correspond to the values $\{\mathbf{tab}(i, j) \mid j \in [i\text{-}\epsilon, i\text{-}1]\}$, $\{\mathbf{tab}(j, i) \mid j \in [i\text{-}\epsilon, i\text{-}1]\}$, and $\mathbf{tab}(i, i)$. The values in these cells greater than ϵ are replaced by $\top$. The initial state of $\mathcal{D}^{\epsilon}_{Lev}$ is $\mathbf{q}_{0\,Lev} = (\lambda, \lambda, \langle \bot, \ldots, \bot, 0, \bot, \ldots, \bot \rangle)$, where λ denotes the empty string, $\bot$ is a special symbol denoting an undefined value, and the value 0 corresponds to entry $\mathbf{tab}(0, 0)$.

Upon reading the i^{th} input symbol pair, the transition of $\mathcal{D}^{\epsilon}_{Lev}$ from state $q_{i\text{-}1}$ to q_i is as shown in Fig. 3.2. Note that to compute values in $\mathbf{e}$ corresponding to the i^{th} row, we need the substring $t[i\text{-}\epsilon, i\text{-}1]$, the values $\mathbf{tab}(i\text{-}1\text{-}\epsilon, i\text{-}1)$ to $\mathbf{tab}(i\text{-}1, i\text{-}1)$, and the symbol s_i. From the invariant on the state, it follows that the values of the required cells from **tab** and the required substring $t[i\text{-}\epsilon, i\text{-}1]$ are present in $q_{i\text{-}1}$ and the input symbol. Similarly, to compute $\mathbf{tab}(j, i)$, where $j \in [i\text{-}1\text{-}\epsilon, i]$ the string in y, values in $\mathbf{e}$ of q_{i-1} and the input symbol suffice. Thus, given any state of $\mathcal{D}^{\epsilon}_{Lev}$ and an input symbol pair, we can construct the unique next state that satisfies the state-invariant.

Recall that for strings s,t, the value of $d_{Lev}(s, t)$ is stored in the entry $\mathbf{tab}(|s|, |t|)$ of **tab**. Keeping this in mind, upon reading the symbol $(\#, \#)$, we add transitions to the accepting state acc_{Lev} iff:

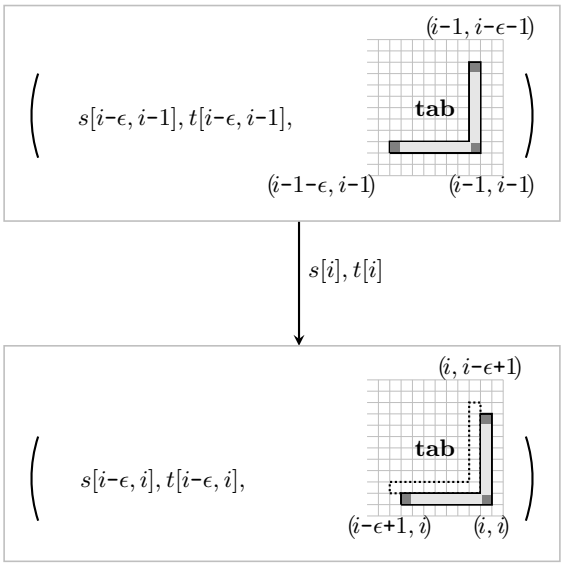


Fig. 3.2. A transition of $\mathcal{D}^{\epsilon}_{Lev}$

- $|s| = |t|$, i.e., x and y do not contain #, and the $(\epsilon+1)^{th}$ entry in $\mathbf{e}$ is $\top$, or,
- $|s| = |t| + \ell$, i.e., y contains ℓ #'s, x contains no #, and the $(\epsilon+1-\ell)^{th}$ entry in $\mathbf{e}$ is $\top$, or,
- $|t| = |s| + \ell$, i.e., x contains ℓ #'s, y contains no #, and the $(\epsilon+1+\ell)^{th}$ entry in $\mathbf{e}$ is $\top$.

This shows how we can construct a $\mathcal{D}^{\epsilon}_{Lev}$ that exactly mimics the dynamic programming algorithm. The following lemma states the correctness of this construction. The proof follows from the state-invariant maintained by $\mathcal{D}^{\epsilon}_{Lev}$ and its acceptance condition.

Lemma 3.2. *$\mathcal{D}^{\epsilon}_{Lev}$ accepts a pair of strings (s,t) iff $d_{Lev}(s,t) > \epsilon$.*

3.3 Automaton for Tracking L_1-norm

The L_1-norm measures the number of positions in which two strings differ. As before, let $s[0] = t[0] = \lambda$. Formally, we define $d_{L_1}(s,t)$ using the following recurrence relations:

$$d_{L_1}(s[0],t[0]) = 0 \qquad d_{L_1}(s[0,j],t[0,j]) = d_{L_1}(s[0,j\text{-}1],t[0,j\text{-}1]) + \mathbf{diff}(s[j],t[j])$$

We now define the automaton $\mathcal{D}^{\epsilon}_{L_1}$ that accepts pairs of strings (s,t) such that $d_{L_1}(s,t) > \epsilon$. The automaton $\mathcal{D}^{\epsilon}_{L_1}$ is a 1-reversal-bounded 1-counter machine (i.e., in NCM(1,1)), defined as a tuple $(\Sigma^{\#} \times \Sigma^{\#}, X_{L_1}, x_{0\,L_1}, Z, G_{L_1}, E_{L_1}, F_{L_1})$, where $(\Sigma^{\#} \times \Sigma^{\#})$ is its input alphabet, $X_{L_1} = \{x_{0\,L_1}, x_{L_1}, acc_{L_1}\}$, is a set of three states, $x_{0\,L_1}$ is the initial state, $Z = \{z\}$ is a single 1-reversal-bounded counter,

$G_{L_1} = \{\epsilon, 0, -1\}$ is a set of integers, and $F_{L_1} = \{acc_{L_1}\}$ is the singleton set of final states. The transition relation contains the following types of transitions:

1. The transition $(x_{0\,L_1}, (\lambda, \lambda), \mathit{true}, x_{L_1}, \epsilon)$ is an initialization transition that sets the counter to ϵ.
2. The transition $(x_{L_1}, (a, a), z \geq 0, x_{L_1}, 0)$ keeps the state and counter of $\mathcal{D}^{\epsilon}_{L_1}$ unchanged upon reading a pair of the same symbols.
3. Transitions of the form $(x_{L_1}, (a, b), z > 0, x_{L_1}, -1)$, for $a \neq b$, decrement the counter by 1 upon reading a pair of distinct symbols. These transitions essentially count the number of differing positions of the two strings.
4. The transition $(x_{L_1}, (a, b), z = 0, acc_{L_1}, 0)$, for $a \neq b$, moves $\mathcal{D}^{\epsilon}_{L_1}$ to an accepting state when it finds the $(\epsilon + 1)^{th}$ differing position. This indicates that the L_1-norm between the strings being read is greater than ϵ.

Lemma 3.3. *$\mathcal{D}^{\epsilon}_{L_1}$ accepts a pair of strings (s, t) iff $d_{L_1}(s, t) > \epsilon$.*

Remark: The construction of $\mathcal{D}^{\epsilon}_{Lev}$ is significantly more involved than that of $\mathcal{D}^{\epsilon}_{L_1}$. This is perhaps clear from the difference in the complexity of the respective recurrence relations. Unlike the L_1-norm, for edit distance computation, it is not sufficient to focus on the positions of edits in each string. One must also obtain the optimal *alignment* or *matching* between strings s and t. For instance, the L_1-norm between the strings *shin* and *hind* is 4, while the edit distance is only 2 (delete s, align/match *hin*, insert d).

4 Analyzing Robustness of a Networked System

In this section, we present an automata-theoretic framework for checking robustness of a networked system in the presence of bounded channel perturbations. Checking if a networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-robust is equivalent to checking if, for each output channel $o_\ell \in O$ (with an error bound of ϵ_ℓ), $\mathcal{N}$is $(\boldsymbol{\delta}, \epsilon_\ell)$-robust. Thus, in what follows, we focus on the problem of checking robustness of the networked system $\mathcal{N}$ for a single output channel. Rephrasing the robustness definition from before, we need to check if for all input strings $\mathbf{s} \in (\Sigma^{|I|})^\star.(\#^{|I|})^\star$, and all runs $\rho_\tau(\mathbf{s})$ of $\mathcal{N}$, $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$ implies that $d([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) \leq \epsilon_\ell$. Here, $[\![\mathcal{N}]\!]|_\ell(\mathbf{s})$, $[\![\rho_\tau]\!]|_\ell(\mathbf{s})$ respectively denote the projections of $[\![\mathcal{N}]\!](\mathbf{s})$ and $[\![\rho_\tau]\!](\mathbf{s})$ on the ℓ^{th} output channel. For simplicity in notation, henceforth, we drop the ℓ in the error bound on the channel, and denote it simply by ϵ.

In what follows, we define composite machines $\mathcal{A}$ that accept input strings certifying the non-robust behavior of a given networked system $\mathcal{N}$. In other words, $\mathcal{A}$ accepts a string $\mathbf{s} \in (\Sigma^{|I|})^\star.(\#^{|I|})^\star$ iff there exists a perturbed execution $\rho_\tau(\mathbf{s})$ of the networked system $\mathcal{N}$ such that: $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$ and $d([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) > \epsilon$. Thus, the networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \boldsymbol{\epsilon})$-robust iff $\mathcal{L}(\mathcal{A})$ is empty.

4.1 Robustness Analysis for the Levenshtein Distance Metric

The composite machine $\mathcal{A}^{\boldsymbol{\delta}, \epsilon}_{Lev}$, certifying non-robustness with respect to the Levenshtein distance metric, is a nondeterministic 1-reversal-bounded $|N|$-counter

machine, i.e., in the class NCM($|N|$,1). In each run on an input string $\mathbf{s}$, $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ simultaneously does the following: (a) it simulates an unperturbed execution $\rho(\mathbf{s})$ and a perturbed execution $\rho_\tau(\mathbf{s})$ of $\mathcal{N}$, (b) keeps track of all the internal channel perturbations along $\rho_\tau(\mathbf{s})$, and (c) tracks the Levenshtein distance between the outputs generated along $\rho(\mathbf{s})$ and $\rho_\tau(\mathbf{s})$.

Similar to the semantics of a networked system $\mathcal{N}$ with multiple output channels, we can define the semantics of $\mathcal{N}$ for the ℓ^{th} output channel using the tuple $(\Sigma^{|I|}, \Sigma, Q, \mathbf{q}_0, R|_\ell)$. Here, $R|_\ell$ denotes the projection of the transition relation R of $\mathcal{N}$ onto the ℓ^{th} output channel. To incorporate the addition of # symbols at the end of strings, the semantics of $\mathcal{N}$ is further modified to the tuple $(\Sigma^{|I|} \cup \{\#^{|I|}\}, \Sigma^{\#}, Q, \mathbf{q}_0, R^{\#})$, where $R^{\#} = R|_\ell \cup \{(\mathbf{q}, ((\#, \ldots, \#), \#), \mathbf{q}) : \mathbf{q} \in Q\}$. Also recall from Sec. 3, that the automaton $\mathcal{D}_{Lev}^{\epsilon}$, accepting pairs of strings with edit distance greater than ϵ from each other, is defined by the tuple $((\Sigma^{\#} \times \Sigma^{\#}), Q_{Lev}, \mathbf{q}_{0\,Lev}, R_{Lev}, F_{Lev})$. Formally, the machine $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$, in the class NCM($|N|$,1), is defined as the tuple $(\Sigma^{|I|} \cup \{\#^{|I|}\}, X, \mathbf{x}_0, Z, G, E, F)$, where $X, \mathbf{x}_0, Z, G, E, F$ are respectively the set of states, initial state, set of counters, a finite set of integers, the transition relation and the final states of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$. We define these below.

The set of states $X = Y \cup \{\mathbf{acc}, \mathbf{rej}\}$, where $Y \subseteq (Q \times Q \times Q_{Lev})$. Each state $\mathbf{x} \in Y$ of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ is a tuple $(\mathbf{q}, \mathbf{r}, \mathbf{q}_{Lev})$, where the component labeled $\mathbf{q}$ tracks the state of $\mathcal{N}$ when it has no channel perturbations, the component $\mathbf{r}$ tracks the state of the network when it has channel perturbations, and $\mathbf{q}_{Lev}$ is a state in $\mathcal{D}_{Lev}^{\epsilon}$.

The initial state of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$, $\mathbf{x}_0$, is given by the tuple $(\mathbf{q}_0, \mathbf{q}_0, \mathbf{q}_{0\,Lev})$. The set of counters $Z = \{z_1, \ldots, z_{|N|}\}$ tracks the number of perturbations in each internal channel of $\mathcal{N}$. The initial value of each counter is 0. $G = \{0, -1, \delta_1, \delta_2, \ldots, \delta_{|N|}\}$ is the set of all integers that can be used in tests on counter values, or by which any counter in Z can be incremented. The set of final states is the singleton set $\{\mathbf{acc}\}$.

The transition relation E of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ is constructed using the following steps:

1. *Initialization transition*:
   From the initial state $\mathbf{x}_0$, we add a single transition of the form:

$$\left((\mathbf{q}_0, \mathbf{q}_0, \mathbf{q}_{0\,Lev}),\ \boldsymbol{\lambda},\ \bigwedge_k z_k = 0,\ (\mathbf{q}_0, \mathbf{q}_0, \mathbf{q}_{0\,Lev}),\ (+\delta_1, \ldots, +\delta_{|N|})\right)$$

   In this transition, $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ sets each counter z_k to the error bound δ_k on the k^{th} internal channel, without consuming an input symbol or changing state. Note that the counter test ensures that this transition can be taken only once from $\mathbf{x}_0$.

2. *Unperturbed network transitions*:
   For all pairs of transitions in the transition relation $R^{\#}$ with the same input symbol, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}')$ and $(\mathbf{r}, \mathbf{a}, b', \mathbf{r}')$, and transitions of the form $(\mathbf{q}_{Lev}, (b, b'), \mathbf{q}'_{Lev}) \in R_{Lev}$, we add a transition of the following form to $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_{Lev}), \quad \mathbf{a}, \quad \bigwedge_k z_k \geq 0, \quad (\mathbf{q}', \mathbf{r}', \mathbf{q}'_{Lev}), \quad \mathbf{0} \right)$$

In each such transition, $\mathcal{A}^{\boldsymbol{\delta},\epsilon}_{Lev}$ consumes an input symbol $\mathbf{a} \in \Sigma^{|I|} \cup \{\#^{|I|}\}$ and simulates a pair of transitions of the unperturbed network $\mathcal{N}$ on $\mathbf{a}$ in the first two components of its state. The distance between the corresponding outputs of $\mathcal{N}$ (b and b' above) is tracked by the third component. Note that in such transitions, all counter values are required to be non-negative in the source state and are not modified.

3. *Perturbed network transitions*:
From each state $\mathbf{x} \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_{Lev}), \quad \boldsymbol{\lambda}, \quad \bigwedge_k z_k \geq 0, \quad (\mathbf{q}, \mathbf{r}_\tau, \mathbf{q}_{Lev}), \quad \mathbf{g} \right)$$

In each such transition, $\mathcal{A}^{\boldsymbol{\delta},\epsilon}_{Lev}$ simulates a τ-transition of the form $(\mathbf{r}, \boldsymbol{\lambda}, \boldsymbol{\lambda}, \mathbf{r}_\tau)$. In the transition, $\mathbf{g}$ denotes a vector with entries in $\{0, -1\}$, where $g_k = -1$ iff the k^{th} internal channel is perturbed in $(\mathbf{r}, \boldsymbol{\lambda}, \boldsymbol{\lambda}, \mathbf{r}_\tau)$. Thus, we model a perturbation on the k^{th} internal channel by decrementing the (nonnegative) z_k counter of $\mathcal{A}^{\boldsymbol{\delta},\epsilon}_{Lev}$. Note that in these transitions, no input symbol is consumed, and the first and third components, i.e. $\mathbf{q}$ and $\mathbf{q}_{Lev}$ remain unchanged.

4. *Rejecting transitions*:
From each state $\mathbf{x} \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_{Lev}), \quad \boldsymbol{\lambda}, \quad \bigvee_k z_k < 0, \quad \mathbf{rej}, \quad \mathbf{0} \right)$$

From the state **rej**, for all $\mathbf{a} \in \Sigma^{|I|}$, we add a transition: $(\mathbf{rej}, \mathbf{a}, \mathit{true}, \mathbf{rej}, \mathbf{0})$.

We add a transition to a designated rejecting state whenever the value of some counter z_k goes below 0, i.e., whenever the perturbation count in some k^{th} internal channel exceeds the error bound $\boldsymbol{\delta}_k$. Once in the state **rej**, $\mathcal{A}^{\boldsymbol{\delta},\epsilon}_{Lev}$ ignores any further input read, and remains in that state.

5. *Accepting transitions*:
Finally, from each state $(\mathbf{q}, \mathbf{r}, \mathit{acc}_{Lev}) \in Y$, we add transitions of the form:

$$\left((\mathbf{q}, \mathbf{r}, \mathbf{q}_{Lev}), \quad \boldsymbol{\lambda}, \quad \bigwedge_k z_k \geq 0, \quad \mathbf{acc}, \quad \mathbf{0} \right)$$

We add a transition to the unique accepting state whenever $\mathbf{q}_{Lev} = \mathit{acc}_{Lev}$ and $\bigwedge_k z_k \geq 0$. The first criterion ensures that $d([\![\mathcal{N}]\!]|_\ell(\mathbf{s}), [\![\rho_\tau]\!]|_\ell(\mathbf{s})) > \epsilon$ (as indicated by reaching the accepting state in $\mathcal{D}^{\epsilon}_{Lev}$). The second criterion ensures that $\|\rho_\tau(\mathbf{s})\| \leq \boldsymbol{\delta}$, i.e., the run $\rho_\tau(\mathbf{s})$ of $\mathcal{N}$ on $\mathbf{s}$ models perturbations on the network that respect the internal channel error bounds.

Theorem 4.1. *Given an upper bound $\boldsymbol{\delta}$ on the number of perturbations in the internal channels, and an upper bound ϵ on the acceptable error for a particular output channel, the problem of checking if the networked system $\mathcal{N}$ is $(\boldsymbol{\delta}, \epsilon)$-robust with respect to the Levenshtein distance is polynomial in the size of $\mathcal{N}$ and exponential in ϵ.*

Proof. We first note that the construction of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ reduces the problem of checking $(\boldsymbol{\delta}, \epsilon)$-robustness of $\mathcal{N}$ (w.r.t. the Levenshtein distance) to checking emptiness of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$. From the construction of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$, we can see that its number of states i.e. $|X|$ is $O(|Q|^2.|Q_{Lev}|)$. From Sec. 3, we know that $|Q_{Lev}|$ is exponential in ϵ. As $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ belongs to the class NCM($|N|$, 1) from Lemma 3.1, we know that checking emptiness of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ is polynomial in the number of states $|X|$ and the number of counters $|N|$ of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$. Thus, checking emptiness of $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ can be done in time polynomial in $|Q|$ and $|N|$ (i.e., the size of the network), and exponential in ϵ.

4.2 Robustness Analysis for the L_1-norm Distance Metric

The composite machine $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$ certifying non-robustness with respect to the L_1-norm metric, is a nondeterministic, 1-reversal-bounded $(|N|+1)$-counter machine, i.e., in the class NCM($|N|+1$, 1). Similar to $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$, the machine $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$ also simultaneously simulates a perturbed and unperturbed execution of the networked system, while tracking the L_1-norm between the outputs generated along both the runs.

Recall from Sec. 3, that the automaton $\mathcal{D}_{L_1}^{\epsilon}$, accepting pairs of strings with L_1-norm greater than ϵ from each other, is in the class NCM(1,1), and is defined by the tuple $(\Sigma^{\#} \times \Sigma^{\#}, X_{L_1}, x_{0\,L_1}, Z, G_{L_1}, E_{L_1}, F_{L_1})$. Formally, the machine $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$, in the class NCM($|N|+1$, is defined as the tuple $(\Sigma^{|I|} \cup \{\#^{|I|}\}, X, \mathbf{x}_0, Z, G, E, F)$, where all components have their usual meaning. The set of states $X = Y \cup \{\mathbf{acc}, \mathbf{rej}\}$, where $Y \subseteq (Q \times Q \times X_{L_1})$. The initial state $\mathbf{x}_0$ is $(\mathbf{q}_0, \mathbf{q}_0, x_{0\,L_1})$. The set of counters $Z = \{z_1, \ldots, z_{|N|}\} \cup \{z\}$, where z is an additional counter used to track the L_1-norm for the output strings. The set $G = \{0, -1, \delta_1, \delta_2, \ldots, \delta_{|N|}, \epsilon\}$, the set F of final states is the singleton set $\{\mathbf{acc}\}$.

We add transitions to E in a step-wise fashion similar to that for $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$:

1. *Initialization transition*:
   We add a single transition of the form:
   $$\left((\mathbf{q}_0, \mathbf{q}_0, x_{0\,L_1}),\ \boldsymbol{\lambda},\ \bigwedge_k z_k = 0 \wedge z = 0,\ (\mathbf{q}_0, \mathbf{q}_0, x_{0\,L_1}),\ (+\delta_1, \ldots, +\delta_{|N|},\ +\epsilon)\right)$$
   In addition to initializing the counters for tracking the internal channel error bounds, this transition also initializes the counter for tracking the L_1-norm of the output strings.

2. *Unperturbed network transitions*:
   For all pairs of transitions in $R^{\#}$ with the same input symbol and output symbol, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}')$ and $(\mathbf{r}, \mathbf{a}, b, \mathbf{r}')$, and transitions of the form

$(x_{L_1}, (b, b), z \geq 0, x_{L_1}, 0)$ *in* E_{L_1}, we add a transition of the following form to $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$:

$$\Big((\mathbf{q}, \mathbf{r}, x_{L_1}), \quad \mathbf{a}, \quad \bigwedge z_k \geq 0 \wedge z \geq 0, \quad (\mathbf{q}', \mathbf{r}', x_{L_1}), \quad (0, \ldots, 0, 0)\Big).$$

For all pairs of transitions in $R^{\#}$ with the same input symbol and different output symbols, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}')$ and $(\mathbf{r}, \mathbf{a}, b', \mathbf{r}')$, and transitions of the form $(x_{L_1}, (b, b'), z > 0, x_{L_1}, -1)$ *in* E_{L_1}, we add a transition of the following form to $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$:

$$\Big((\mathbf{q}, \mathbf{r}, x_{L_1}), \quad \mathbf{a}, \quad \bigwedge z_k \geq 0 \wedge z > 0, \quad (\mathbf{q}', \mathbf{r}', x_{L_1}), \quad (0, \ldots, 0, -1)\Big).$$

For all pairs of transitions in $R^{\#}$ with the same input symbol and different output symbols, i.e., $(\mathbf{q}, \mathbf{a}, b, \mathbf{q}')$ and $(\mathbf{r}, \mathbf{a}, b', \mathbf{r}')$, and transitions of the form $(x_{L_1}, (b, b'), z = 0, acc_{L_1}, 0)$ *in* E_{L_1}, we add transitions of the following form to $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$:

$$\Big((\mathbf{q}, \mathbf{r}, x_{L_1}), \quad \mathbf{a}, \quad \bigwedge z_k \geq 0 \wedge z = 0, \quad (\mathbf{q}', \mathbf{r}', acc_{L_1}), \quad (0, \ldots, 0, 0)\Big).$$

The perturbed network transitions, rejecting transitions, and accepting transitions are added in a similar fashion to $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$, (substitute $\mathbf{q}_{Lev}$ in all transitions for $\mathcal{A}_{Lev}^{\boldsymbol{\delta},\epsilon}$ by x_{L_1}).

Theorem 4.2. *Given an upper bound* $\boldsymbol{\delta}$ *on the number of perturbations in the internal channels, and an upper bound* ϵ *on the acceptable error for a particular output channel, the problem of checking if the networked system* $\mathcal{N}$ *is* $(\boldsymbol{\delta}, \epsilon)$*-robust with respect to the* L_1*-norm is polynomial in the size of the network* $\mathcal{N}$*.*

Proof. We note that the construction of $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$ reduces the problem of checking $(\boldsymbol{\delta}, \epsilon)$-robustness for $\mathcal{N}$ (w.r.t. the L_1-norm) to checking emptiness of $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$. As $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$ belongs to the class NCM($|N|$+1,1), from Lemma 3.1, we know that checking emptiness of $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$ is polynomial in $|X|$ (number of states of $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$) and $(|N| + 1)$. From the construction of $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$, we know that $|X| = O(|Q|^2.|Q_{L_1}|)$, and from Sec. 3, we know that $|Q_{L_1}|$ is a small constant. Thus, checking emptiness of $\mathcal{A}_{L_1}^{\boldsymbol{\delta},\epsilon}$ can be done in time polynomial in $|Q|$ and $|N|$, i.e., polynomial in the size of the network.

5 Related Work

There is a growing interest in the study of robustness in the formal methods and software engineering communities. The initial papers by Majumdar and Saha [17] and by Chaudhuri et al [5,6,7] consider robustness of infinite-state programs. The programs considered in these papers are essentially functional; their scope does not extend to concurrent systems with channel errors like ours.

More recent papers have aimed to develop a notion of robustness for reactive systems. In [22], the authors propose a comprehensive notion of input-output stability of finite-state transducers that bounds both the deviation of the output from disturbance-free behaviour under bounded disturbance, as well as the persistence of the effect on the output of a sporadic disturbance. The deviations are measured using cost functions that map strings to nonnegative integers. The authors present polynomial-time algorithms for the analysis and synthesis of robust transducers. Exploring extensions of techniques presented in our paper to address persistence of a sporadic disturbance would be interesting.

In [18,4,2], the authors develop different notions of robustness for reactive systems, with ω-regular specifications, interacting with uncertain environments. In [18], the authors present metric automata, which are automata equipped with a metric on states. The authors assume that at any step, the environment can perturb any state q to a state at most $\gamma(q)$ distance away, where γ is some function mapping states to real numbers. A winning strategy for a finite-state or Büchi automaton $\mathcal{A}$ is a strategy that satisfies the corresponding acceptance condition (stated as reachability of states in F or as infinitely often visiting states in F respectively). Such a winning strategy is defined to be σ-robust if it is a winning strategy for $\mathcal{A}$ where the set F' characterizing the acceptance condition includes all states at most $\sigma.\sup_{q \in F}\gamma(q)$ distance away from the F. We note that while there are some similarities in how a disturbance is modeled, our approach is quite different, as we quantify and analyze the effect of errors over time, and do not associate metrics with individual states.

In [8], the authors study robustness of sequential circuits w.r.t. a *common suffix distance* metric. Their notion of robustness essentially bounds the persistence of the effect of a sporadic disturbance in the input of a sequential circuit. To be precise, a circuit is said to be robust iff the position of the last mismatch in any pair of output sequences is a bounded number of positions from the last mismatch position in the corresponding pair of input sequences. The authors present a polynomial-time algorithm to decide robustness of sequential circuits modeled as (single) Mealy machines. The metric and its subsequent treatment developed in this paper is useful for analyzing circuits; however, for networked systems communicating via strings, metrics such as edit distance and the L_1-norm provide a more standard way to measure the effect of errors.

In [9], the authors present modeling techniques for cyber-physical systems. Further, the authors also discuss the challenges of including a network in a cyber-physical system. A key observation is that to maintain discrete-event semantics of components in such a system, it is important to have a common sense of time across all components. A critical requirement in such systems is that the communication remain synchronized, which is typically fulfilled by using protocols that bound the allowed drift in the value of the global clock. In our model, we do not analyze such details, and abstract them away, assuming that some underlying protocol ensures synchronous communication.

Work in the area of robust control seeks to analyze and design networked control systems where communication between sensors, the controller, and

actuators occurs over unreliable networks such as wireless networks [1]. On the other hand, work on wireless control networks [20,21] focuses on design of distributed controllers where the components of the controller communicate over unreliable wireless networks. In such applications, robustness typically means desirable properties of the control loop such as stability. We note that these papers typically assume a synchronous communication schedule as supported by wireless industrial control protocols such as ISA 100 and WirelessHART.

6 Discussion

We have presented a framework for the analysis of robustness of networked systems in the presence of bounded channel perturbations. There are a few directions in which this framework can be developed further. The first is a more extensive treatment of distance metrics. We observe that the symbol sequences (in Σ^*) in a networked cyber-physical system could represent a wide range of digital signals. To accurately model the deviation of such signals in an error-prone network from their error-free counterparts, one must track the *magnitude* of the signals. This necessitates defining and computing distances that are based on mapping individual symbols or symbol sequences to numbers [22].

The second direction is a generalization of the error model and subsequently, the robustness definition. In this work, we only focus on internal channel errors in a network, and assume that the input and output channels are error-free. However, in a real-world scenario, there can be multiple sources of uncertainty such as sensor and actuator noise, modeling errors and process failures. A comprehensive robustness analysis should thus check if small disturbances in the inputs or internal channels or processes result in small deviations in the system behaviour.

Finally, we also wish to investigate the extension of our current techniques to the *design* of robust networks.

References

1. Alur, R., D'Innocenzo, A., Johansson, K.H., Pappas, G.J., Weiss, G.: Compositional Modeling and Analysis of Multi-Hop Control Networks. IEEE Transactions on Automatic Control 56(10), 2345–2357 (2011)
2. Bloem, R., Greimel, K., Henzinger, T., Jobstmann, B.: Synthesizing Robust Systems. In: Proceedings of Formal Methods in Computer Aided Design (FMCAD), pp. 85–92 (2009)
3. Boussinot, F., De Simone, R.: The ESTEREL language. Proceedings of the IEEE 79(9), 1293–1304 (1991)
4. Černý, P., Henzinger, T.A., Radhakrishna, A.: Simulation Distances. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 253–268. Springer, Heidelberg (2010)
5. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity Analysis of Programs. In: Proceedings of Principles of Programming Languages (POPL), pp. 57–70 (2010)

6. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and Robustness of Programs. Communications of the ACM (2012)
7. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving Programs Robust. In: Proceedings of Foundations of Software Engineering, pp. 102–112 (2011)
8. Doyen, L., Henzinger, T.A., Legay, A., Ničković, D.: Robustness of Sequential Circuits. In: Proceedings of Application of Concurrency to System Design (ACSD), pp. 77–84 (2010)
9. Eidson, J.C., Lee, E.A., Matic, S., Seshia, S.A., Zou, J.: Distributed Real-Time Software for Cyber-Physical Systems. Proceedings of the IEEE (Special Issue on CPS) 100(1), 45–59 (2012)
10. Eilenberg, S.: Automata, Languages, and Machines, vol. A. Academic Press, New York (1974)
11. Frougny, C., Sakarovitch, J.: Rational Relations with Bounded Delay. In: Jantzen, M., Choffrut, C. (eds.) STACS 1991. LNCS, vol. 480, pp. 50–63. Springer, Heidelberg (1991)
12. Gurari, E.M., Ibarra, O.H.: The Complexity of Decision Problems for Finite-Turn Multicounter Machines. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 495–505. Springer, Heidelberg (1981)
13. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
14. Ibarra, O.H.: Reversal-Bounded Multicounter Machines and Their Decision Problems. Journal of the ACM 25(1), 116–133 (1978)
15. Ibarra, O.H., Su, J., Dang, Z., Bultan, T., Kemmerer, R.A.: Counter Machines: Decidable Properties and Applications to Verification Problems. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, pp. 426–435. Springer, Heidelberg (2000)
16. Kahn, G.: The Semantics of Simple Language for Parallel Programming. In: IFIP Congress, pp. 471–475 (1974)
17. Majumdar, R., Saha, I.: Symbolic Robustness Analysis. In: 30th IEEE Real-Time Systems Symposium, pp. 355–363 (2009)
18. Majumdar, R., Render, E., Tabuada, P.: A Theory of Robust Software Synthesis. CoRR abs/1108.3540 (2011)
19. Mealy, G.H.: A Method for Synthesizing Sequential Circuits. Bell Systems Technical Journal, 1045–1079 (1955)
20. Pajic, M., Sundaram, S., Pappas, G.J., Mangharam, R.: The Wireless Control Network: A New Approach for Control Over Networks. IEEE Transactions on Automatic Control 56(10), 2305–2318 (2011)
21. Pappas, G.J.: Wireless Control Networks: Modeling, Synthesis, Robustness, Security. In: Proceedings of Hybrid Systems: Computation and Control (HSCC), pp. 1–2 (2011)
22. Tabuada, P., Balkan, A., Caliskan, S.Y., Shoukry, Y., Majumdar, R.: Input Output Stability for Discrete Systems. In: Proceedings of International Conference on Embedded Software, EMSOFT (2012)

Causality Checking for Complex System Models

Florian Leitner-Fischer and Stefan Leue

University of Konstanz, Germany

Abstract. We present an approach for the algorithmic computation of causalities in system models that we refer to as *causality checking*. We base our notion of causality on counterfactual reasoning, in particular using the structural equation model approach by Halpern and Pearl that we recently have extended to reason about computational models. In this paper we present a search-based on-the-fly approach that nicely integrates into finite state verification techniques, such as explicit-state model checking. We demonstrate the applicability of our approach using an industrial case study.

1 Introduction

Model Checking [1] is an established technique for the automated analysis of system properties. If a model of the system and a formalized property is given to the model checker, it automatically checks whether it can find property violations. In case some property is violated, the model checker returns a counterexample, which consists of a system execution trace leading to the property violation. While a counterexample helps in retracing the system execution leading to the property violation, it does not identify causes of the property violation.

We present an approach based on explicit state space search towards the automated computation of causalities that we refer to as *causality checking*. Instead of returning just a single counterexample at the end of the model checking process, we compute causal events that lead to the violation of a desired system property. The notion of causality that we use is based on counterfactual reasoning [2,3].

In precursory work [4] causality computation was performed as a postprocessing step on a set of probabilistic counterexamples. In addition we presented a mapping of the computed causality relationships between events to fault trees. For the causality computation all possible execution traces need to be computed and stored on disk prior to the causality checking. The current paper focuses on a extension of our causality model and an integration of the causality computation into standard state-space search as used by explicit-state model checkers. Consequently, it is no longer necessary to store all good and bad execution traces before performing the causality computation. We tailor the causality model from [4] so that it can be used for the analysis of concurrent system models described by transition systems. We also show how the causality checks can be mapped to sub- and superset comparisons of execution traces. The proposed algorithm for causality checking is an extension of the depth-first search and breadth-first

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 248–267, 2013.

search algorithms used for state-space exploration in explicit-state model checking. In keeping with standard practice in this domain we design our algorithms to work on-the-fly. To his end we propose a data-structure called subset graph that is used to store the counterexamples that are needed for causality checking. A further contribution of our current paper is an application of this approach to two case studies, one of them of industrial size, and a comparison of various search strategies.

The remainder of this paper is structured as follows. In Section 2 we discuss how causality relationships can be formally established within system models. The on-the-fly algorithm for causality computation and its integration in state-space exploration algorithms is presented in Section 3. In Section 4 we demonstrate the causality checking approach using two case studies. Related work is discussed throughout the paper and in Section 5. We conclude in Section 6.

2 Causality Reasoning in System Models

Our goal is to identify the events that cause the violation of a non-reachablitiy requirement. Such a violation could, for instance, represent a hazard or a potentially unsafe state of the system. We use the explicit state model checker SPIN [5] to check whether there are system executions that lead to such an undesired state.

2.1 System Model

The systems that we wish to apply causality checking to are concurrent systems. For the formalization of the system model we follow the formalization of a model for concurrent computing systems proposed in [6]. The system model is given by a Transition System which is defined as follows:

Definition 1. *Transition System. A* transition system *TS is a tuple* $(S, Act, \rightarrow, I, AP, L)$ *where* S *is a finite set of* states*,* Act *is a finite set of* actions*,* $\rightarrow \subseteq S \times Act \times S$ *is a* transition relation*,* $I \subseteq S$ *is a set of* initial states*,* AP *is a set of* atomic propositions*, and* $L : S \rightarrow 2^{AP}$ *is a* labeling function*.*

A Transition System defines a Kripke structure. Each state $s \in S$ is labeled with the set $L(s)$ of all atomic state propositions that are true in this state. The set Act contains all actions that can trigger the system to transit from some state into a successor state. The execution semantics of a transition system is defined as follows:

Definition 2. *Execution Trace of a Transition System. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *be a transition system. A finite execution* σ *of* T *is an alternating sequence of states* $s \in S$ *and actions* $\alpha \in Act$ *ending with a state.* $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ \ldots\ \alpha_n\ s_n$ *s.t.* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ *for all* $0 \leq i < n$*.*

The analysis aims at identifying the violation of functional safety requirements. Such a violation is also referred to as a hazard. We use linear time temporal

logic (LTL) using its standard syntax and semantics as defined in [7] in order to specify hazards. Hazards imply the reachability of unsafe states and they hence belong to the class of reachability properties. Hence we only need to consider finite execution fragments [6]. Hazards fall within the class of safety properties in the commonly used classification scheme of safety and liveness properties. We use $T \vDash_l \varphi$ to express that the LTL formula φ holds for the transition system T and $\sigma \vDash_l \varphi$ respectively for execution traces.

We will demonstrate the presented definitions on a running example of a railroad crossing system. In the running example a train can approach the crossing (Ta), cross the crossing (Tc) and finally leave the crossing (Tl). Whenever a train is approaching, the gate should close (Gc) and will open when the train has left the crossing (Go). It might also be the case that the gate fails (Gf). The car approaches the crossing (Ca) and crosses the crossing (Cc) if the gate is open and finally leaves the crossing (Cl). We are interested in finding those events that lead to a hazard state in which both the car and the train are in the crossing. This hazard can be characterized by the LTL formula $\varphi = \square\neg(car\_crossing \wedge train\_crossing)$.

In the following we will use short-hand notation σ = "a_{α_1},a_{α_2}, ... , a_{α_n}" for an execution trace $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ ...\ \alpha_n\ s_n$. The trace σ = "Ta, Ca, Gf, Cc, Tc", for instance, is a trace of the railroad example where the train and the car are approaching the crossing (Ta, Ca), the gate fails to close (Gf), the car crosses the crossing (Cc) and finally the train crosses the crossing (Tc).

We can partition the set of all possible execution traces Σ of a transition system T into the set of "good" execution traces, denoted Σ_G, where the LTL formula is not violated and thus the hazard does not occur, and the set of "bad" execution traces, denoted Σ_B, where the LTL formula is violated and thus the hazard occurs. The elements of Σ_B are also referred to as counterexamples in model checking. The trace σ = "Ta, Ca, Gf, Cc, Tc" we already discussed above is a "bad" execution trace, since bot the car and the train are on the crossing at the same time and thus the LTL property is violated. An example for a "good" trace is σ' = "Ta, Ca, Gf, Cc, Cl, Tc" where the car leaves the crossing (Cl) before the train is crossing (Tc) and consequently the train and the car are not on the crossing at the same time and the LTL formula is not violated.

Definition 3. *Good and Bad Execution Traces. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *be a transition system, let* φ *an LTL formula over AP and* Σ *that set of all possible finite executions of T. The set* Σ *is divided into into the set of "good" execution traces* Σ_G *and in the set of "bad" execution traces* Σ_B *as follows:* $\Sigma_G = \{\sigma \in \Sigma \mid \sigma \vDash_l \varphi\}$, $\Sigma_B = \{\sigma \in \Sigma \mid \sigma \nvDash_l \varphi\}$ *and* $\Sigma_G \cup \Sigma_B = \Sigma$ *and* $\Sigma_G \cap \Sigma_B = \varnothing$.

2.2 Causality Reasoning

Our goal is to automatically identify those events that are causal for the violation of an LTL property. We assume that for a given execution trace σ of a transition system T, Act contains the events that we wish to reason about. For an LTL formula φ specifying a safety requirement and an execution trace σ, the hazard

described by the safety requirement occurs on σ if and only if $\sigma \not\models_l \varphi$ holds. Notice that since each transition is only labeled with one action, only one event can occur per transition. In order to be able to reason about the causality of events we have to formally capture the occurrence of events. We assume that there exists a set $\mathcal{A}$ of event variables that contains a boolean variable a for each action $\alpha \in Act$ for some given transition system. The variable a_{Ta} for instance represents the event train approaching the crossing. If multiple instances of one event type occur on one execution trace, for example the two train approaching events on "Ta,Gc,Tc,Tl,Go,Ta", the variables representing them are numbered according to their occurrence, for our example a_{Ta_1} and a_{Ta_2}. In other words, the *i-th* occurrence of some action of type α will be represented by the boolean variable a_{α_i}. In the following we also abbreviate the event variable a_{Ta} by Ta.

Definition 4. *Events, Event Types and Event Variables. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system and* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a finite execution trace of T. We define the following: each* $\alpha \in Act$ *defines an* event type α. α_i *of* σ *denotes the* i-th *occurrence of an event of the* event type α. *The variable representing the occurrence of the event* α_i *is denoted by* a_{α_i}, *and the set* $\mathcal{A} = \{a_{\alpha_1}, ..., a_{\alpha_n}\}$ *contains a boolean variable for each occurrence of an event.*

Event variables allow us to reason about the occurrence of single events, but since we want to reason about the combination of events, we need a formalism that allows us to express the occurrence of event combinations. In [4] we presented the event order logic (EOL) which allows one to connect event variables from $\mathcal{A}$ with the boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events we introduced the ordered conjunction operator $\barwedge$. The formula $a \barwedge b$ with $a, b \in \mathcal{A}$ is satisfied if and only if events a and b occur in a trace and a occurs before b. We present here an amended version of the event order logic and further refine it in order to enable causality reasoning for concurrent system models specified by transition systems. In addition to the $\barwedge$ operator we introduce the interval operators $\barwedge_{[}$, $\barwedge_{]}$, and $\barwedge_{<} \phi \barwedge_{>}$, which define an interval in which an event has to hold in all states. As we will see later, these interval operators are necessary to express the causal non-occurrence of events.

Definition 5. *Syntax of Event Order Logic (EOL). Simple event order logic formulas over the set* $\mathcal{A}$ *of event variables are formed according to the following grammar:*

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 \vee \phi_2$$

where $a \in \mathcal{A}$ *and* ϕ, ϕ_1 *and* ϕ_2 *are simple event order logic formulas. Complex event order logic formulas are formed according to the following grammar:*

$$\psi ::= \phi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \barwedge \psi_2 \mid \psi \barwedge_{[} \phi \mid \phi \barwedge_{]} \psi \mid \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2$$

where ϕ *is a simple event order logic formula and* ψ_1 *and* ψ_2 *are complex event order logic formulas. Note that the* $\neg$ *operator binds more tightly than the* $\barwedge$, $\barwedge_{[}$, $\barwedge_{]}$, *and* $\barwedge_{<} \phi \barwedge_{>}$, *operators and those bind more tightly than the* $\vee$ *and* $\wedge$ *operator.*

The formal semantics of this logic is defined on execution traces. Notice that the $\wedge$, $\wedge_{[}$, $\wedge_{]}$, and $\wedge_{<}\ \phi\ \wedge_{>}$ operators are linear temporal logic operators and that the execution trace σ is akin to a linearly ordered Kripke structure.

Definition 6. *Semantics of Event Order Logic (EOL). Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, let* ϕ, ϕ_1, ϕ_2 *simple event order logic formulas, let* ψ, ψ_1, ψ_2 *complex event order logic formulas, and let* $\mathcal{A}$ *a set of event variables, with* $a_{\alpha_i} \in \mathcal{A}$, *over which* ϕ, ϕ_1, ϕ_2 *are built. Let* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a finite execution trace of* T *and* $\sigma[i..r] = s_i, \alpha_{i+1}, s_{i+1}, \alpha_{i+2}, \ldots \alpha_r, s_r$ *a partial trace. We define that an execution trace* σ *satisfies a formula* ψ, *written as* $\sigma \vDash_e \psi$, *as follows:*

- $s_j \vDash_e a_{\alpha_i}$ iff $s_{j-1} \xrightarrow{\alpha_i} s_j$
- $s_j \vDash_e \neg\phi$ iff not $s_j \vDash_e \phi$
- $\sigma[i..r] \vDash_e \phi$ iff $\exists j : i \leq j \leq r\ .\ s_j \vDash_e \phi$
- $\sigma \vDash_e \psi$ iff $\sigma[0..n] \vDash_e \psi$, *where* n *is the length of* σ.
- $\sigma[i..r] \vDash_e \phi_1 \wedge \phi_2$ iff $\sigma[i..r] \vDash_e \phi_1$ *and* $\sigma[i..r] \vDash_e \phi_2$
- $\sigma[i..r] \vDash_e \phi_1 \vee \phi_2$ iff $\sigma[i..r] \vDash_e \phi_1$ *or* $\sigma[i..r] \vDash_e \phi_2$
- $\sigma[i..r] \vDash_e \psi_1 \wedge \psi_2$ iff $\sigma[i..r] \vDash_e \psi_1$ *and* $\sigma[i..r] \vDash_e \psi_2$
- $\sigma[i..r] \vDash_e \psi_1 \vee \psi_2$ iff $\sigma[i..r] \vDash_e \psi_1$ *or* $\sigma[i..r] \vDash_e \psi_2$
- $\sigma[i..r] \vDash_e \psi_1 \wedge \psi_2$ iff $\exists j, k : i \leq j < k \leq r\ .\ \sigma[i..j] \vDash_e \psi_1$ *and* $\sigma[k..r] \vDash_e \psi_2$
- $\sigma[i..r] \vDash_e \psi \wedge_{[} \phi$ iff $(\exists j : i \leq j \leq r\ .\ \sigma[i..j] \vDash_e \psi$ *and* $(\forall k : j \leq k \leq r\ .\ \sigma[k..k] \vDash_e \phi))$
- $\sigma[i..r] \vDash_e \phi \wedge_{]} \psi$ iff $(\exists j : i \leq j \leq r\ .\ \sigma[j..r] \vDash_e \psi$ *and* $(\forall k : 0 \leq k \leq j\ .\ \sigma[k..k] \vDash_e \phi))$
- $\sigma[i..r] \vDash_e \psi_1 \wedge_{<} \phi \wedge_{>} \psi_2$ iff $(\exists j, k : i \leq j < k \leq r\ .\ \sigma[i..j] \vDash_e \psi_1$ *and* $\sigma[k..r] \vDash_e \psi_2$ *and* $(\forall l : j \leq l \leq k\ .\ \sigma[l..l] \vDash_e \phi))$

We define that the transition system T *satisfies the formula* ψ, *written as* $T \vDash_e \psi$, *iff* $\exists \sigma \in T\ .\ \sigma \vDash_e \psi$.

Each execution trace σ specifies an assignment of the boolean values *true* and *false* to the variables in the set $\mathcal{A}$. If an event α_i occurs on σ its value is set to *true*. If the event does not occur on σ its value is set to *false*. We define a function $val_{\mathcal{A}}(\sigma)$ that represents the valuation of all variables in $\mathcal{A}$ for a given σ.

Definition 7. *Valuation of the Set of Event Variables. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system,* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a finite execution trace of* T *and* $\mathcal{A}$ *the set of event variables then we define the function* $val_{\mathcal{A}}(\sigma)$ *as follows:*

$$val_{\mathcal{A}}(\sigma) = (a_{\alpha_1}, ..., a_{\alpha_n}) \mid a_{\alpha_i} = \begin{cases} \text{true } \textit{if } \sigma \vDash_e a_{\alpha_i} \\ \text{false, } \textit{else} \end{cases}.$$

Further we define $val_{\mathcal{A}}(\sigma) = val_{\mathcal{A}}(\sigma')$ *if for all* $a_{\alpha_i} \in \mathcal{A}$ *the values assigned by* $val_{\mathcal{A}}(\sigma)$ *and* $val_{\mathcal{A}}(\sigma')$ *are equal and* $val_{\mathcal{A}}(\sigma) \neq val_{\mathcal{A}}(\sigma')$ *else.*

In fact, we can represent an execution trace by an EOL formula. Suppose we want to represent the execution trace σ = "Ta, Ca, Gf, Cc, Tc" by an EOL formula. We partition the set $\mathcal{A}$ of event variables in the set Z containing all the event variables of the events that occur on σ and the set W containing all the event variables of the events that do not occur on σ. Consequently, Z contains Ta, Ca, Gf, Cc, and Tc. The resulting EOL formula over Z is $\psi = \mathrm{Ta} \wedge \mathrm{Ca} \wedge \mathrm{Gf} \wedge \mathrm{Cc} \wedge \mathrm{Tc}$.

Definition 8. *Event Order Logic (EOL) Formula over Executions. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ \ldots\ \alpha_n\ s_n$ *an execution trace of* T*. The EOL over the execution* σ *denoted by* ψ_σ *is defined as follows: We partition the set of event variables* $\mathcal{A}$ *into sets* Z *and* W *in such a way that* Z *contains all event variables of the events that occur on* σ *and* W *contains all event variables of the events that do not occur on* σ*.* ψ_σ *is the EOL formula containing all events in* Z *in the order they occur on* σ *(e.g.* $\psi_\sigma = a_{\alpha_1} \wedge a_{\alpha_2} \wedge \ldots \wedge a_{\alpha_n}$*).*

Now that we have established the formal basis to reason about the occurrence of events we have to formally define the notion of causality that we will use. A commonly adopted notion of causality is that of *counterfactual* reasoning and the related *alternative world* semantics of Lewis [2,8]. The "naive" counterfactual causality criterion according to Lewis is as follows: event A is causal for the occurrence of event B if and only if, were A not to happen, B would not occur. The testing of this condition hinges upon the availability of alternative worlds. A causality can be inferred if there is a world in which A and B occur, whereas in an alternative world neither A nor B occurs. In our setting possible system execution traces represent the alternative worlds.

The *structural equation model (SEM)* by Halpern and Pearl [3] extends the counterfactual reasoning approach by Lewis. The SEM introduces the notion of causes being logical combinations of events as well as a distinction of relevant and irrelevant causes. In the SEM events are represented by variable values and the minimal number of causal variable valuation combinations is determined. In order to do so the counterfactual test is extended by contingencies. Contingencies can be viewed as possible alternative worlds, where a variable value is changed. A variable X is causal if there exists a contingency, that is a variable valuation for other variables, that makes X counterfactual. In our precursory work [4], we extended the SEM by considering the order of the occurrences of events as possible causal factors. We now present an adaption of the SEM that can be used to decide whether a given EOL formula ψ describes the causal process of the violation of some LTL formula φ in a transition system. The causal process [3] comprises the causal events for the property violation and all events that mediate between the causal events and the property violation. Those events which are not root causes, are needed to propagate the cause through the system until the property violation is being triggered. If ψ describes the causal process of a property violation we also say ψ is causal for the property violation.

In a naive causality checking algorithm we perform the tests defined in Definition 9 for the induced EOL formula ψ_σ of each $\sigma \in \Sigma_B$. The disjunction of

all $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_n}$ that satisfy the conditions AC1-AC3 is the EOL formula describing all possible causes of the hazard.

Definition 9. *Cause for a Property Violation (Adapted SEM). Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* σ, σ' *and* σ'' *some execution traces of* T. *We partition the set of event variables* $\mathcal{A}$ *into sets* Z *and* W. *An EOL formula* ψ *consisting of the event variables in* Z *is considered a cause for an effect represented by the violation of the LTL formula* φ, *if the following conditions are satisfied:*

- *AC1: There exists an execution* σ, *for which both* $\sigma \vDash_e \psi$ *and* $\sigma \nvDash_l \varphi$ *hold.*
- *AC2 (1):* $\exists \sigma'$ *s.t.* $\sigma' \nvDash_e \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ *and* $\sigma' \vDash_l \varphi$. *In words, there exists an execution* σ' *where the order and occurrence of events is different from execution* σ *and* φ *is not violated on* σ'.
- *AC2 (2):* $\forall \sigma''$ *with* $\sigma'' \vDash_e \psi \wedge (val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma''))$ *it holds that* $\sigma'' \nvDash_l \varphi$ *for all subsets of* W. *In words, for all executions where the events in* Z *have the value defined by* $val_Z(\sigma)$ *and the order defined by* ψ, *the value and order of an arbitrary subset of the events in* W *have no effect on the violation of* φ.
- *AC3: The EOL formula* ψ *is minimal: no subset of* ψ *satisfies conditions AC1 and AC2.*

If we want, for instance, to show that $\psi = \text{Ta} \wedge \text{Ca} \wedge \text{Gf} \wedge \text{Cc} \wedge \text{Tc}$ is causal, we need to show that AC1, AC2(1), AC2(2) and AC3 are fulfilled for ψ.

- AC1 is fulfilled, since there exists an execution σ = "Ta, Ca, Gf, Cc, Tc" for which $\sigma \vDash_e \psi$, and both the train and the car are in the crossing at the same time.
- AC2(1) is fulfilled since there exists an execution σ' = "Ta, Ca, Gc, Tc" for which $\sigma' \nvDash_e \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \wedge val_W(\sigma) \neq val_W(\sigma'))$ holds and σ' does not violate the property.
- Now we need to check the condition AC2(2). For the execution σ'' ="Ta, Ca, Gf, Cc, Cl, Tc" and the partition $Z, W \subseteq \mathcal{A}$, $\sigma'' \vDash_e \psi$ and $val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma'')$ hold. The property is not violated since the car leaves the crossing (Cl) before the train enters the crossing (Tc). As a consequence, AC2(2) is not fulfilled by ψ because if Cl occurs between Cc and Tc, the property violation is prevented.

The example showed that the non-occurrence of events can be causal as well, and that this is not yet captured by the adapted SEM. The non-occurrence of an event is causal when ever AC1 and AC2(1) are fulfilled but AC2(2) fails for a EOL formula ψ_σ. If AC2(2) fails there is at least one event α on σ'' which did not occur on σ and the occurrence of α prevents the property violation. Consequently, the non-occurrence of α on σ is causal. We need to reflect the causal effect of the non-occurrence of α in ψ_σ. For the models that we analyze there are three possibilities for such a preventing event α to occur, namely, at the beginning of the execution trace, at the end of the execution trace, or between two other events α_1 and α_2. Furthermore, it is possible that the property

violation is prevented by more than one event, hence we need to find the minimal set of events that are needed to prevent the property violation. This is achieved by finding the minimal true subset $Q \subset W$ of event variables that need to be changed in order to prevent the property violation.

Definition 10. *Non-Occurrence of Events. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* σ *and* σ'' *execution traces of* T*. We partition the set of event variables* $\mathcal{A}$ *into sets* Z *and* W*. Let* ψ *an EOL formula consisting of the event variables in* Z*. The non-occurrence of the events which are represented by the event variables* $a_\alpha \in Q$ *with* $Q \subseteq W$ *on execution* σ *is causal for the violation of the LTL formula* φ *if* ψ *satisfies AC1 and AC2(1) but violates AC2(2), and if* Q *is minimal, which means that there is no true subset of* Q *for which* $\sigma'' \vDash_e \psi \wedge val_Z(\sigma) = val_Z(\sigma'') \wedge val_Q(\sigma) \neq val_Q(\sigma'') \wedge val_{W \setminus Q}(\sigma) = val_{W \setminus Q}(\sigma'')$ *and* $\sigma'' \nvDash_l \varphi$ *holds.*

For each event variable $a_\alpha \in Q$ we determine the location of the event in ψ'' and prohibit the occurrence of α in the same location in ψ. We add $\neg a_\alpha \barwedge_]$ at the beginning of ψ if the event occurred at the beginning of σ'' and $\barwedge_[\neg a_\alpha$ at the end of ψ if the event occurred at the end of σ''. If α occurred between the two events α_1 and α_2 we insert $\barwedge_< \neg a_\alpha \barwedge_>$ between the two event variables a_{α_1} and a_{α_2} in ψ. Additionally, each event variable in Q is added to Z. In our example, Cl is the only event that can prevent the property violation on σ and occurs between the events Cc and Tc. Consequently $\neg Cl$ is added to Z and ψ and we get $\psi = \mathrm{Ta} \barwedge \mathrm{Ca} \barwedge \mathrm{Gf} \barwedge \mathrm{Cc} \barwedge_< \neg \mathrm{Cl} \barwedge_> \mathrm{Tc}$.

If a formula ψ meets conditions AC1 through AC3, the occurrence of the events included in ψ is causal for the violation of φ. However, condition AC2 does not imply that the order of the occurring events is causal. For instance, we do not know whether Ta occurring before Ca is causal in our example or not. If the order of the events is not causal, then there has to exists an execution for each ordering of the events that is possible in the system, and these executions all violate the property. Whether the order of events is causal is checked by the following Order Condition (OC1). Note that the outcome of OC1 has no effect on ψ being causal, but merely indicates whether in addition the order of events in ψ is causal.

Definition 11. *Order Condition (OC1). Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* σ*,* σ' *execution traces of* T*. Let* ψ *an EOL formula over* Z *that holds for* σ *and let* $\psi_\wedge$ *the EOL formula that is created by replacing all* $\barwedge$*-operators in* ψ *by* $\wedge$*-operators. The* $\barwedge_[$*,* $\barwedge_]$*, and* $\barwedge_< \phi \barwedge_>$ *are not replaced in* $\psi_\wedge$*.*

OC1: The order of a subset of events $Y \subseteq Z$ *represented by the EOL formula* χ *is not causal if the following holds:* $\sigma \vDash_e \chi \wedge \exists \sigma' \in \Sigma_B : \sigma' \nvDash_e \chi \wedge \sigma' \vDash_e \chi_\wedge$*.*

In our example, the order of the events Gf, Cc, ¬Cl, Tc is causal since only if the gate fails before the car and the train are entering the crossing, and the car does not leave the crossing before the train is entering the crossing an accident happens. Consequently after OC1 we obtain the EOL formula $\psi = \mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_< \neg \mathrm{Cl} \barwedge_> \mathrm{Tc})$.

3 On-The-Fly Causality Checking

3.1 Preliminaries

In order to compute causality relationships, it is necessary to compute good and bad execution traces. If depth-first search or breadth-first search is used for model checking, good and bad executions can easily be retrieved by the counterexample reporting capabilities of the model checker in use.

The key idea of the proposed algorithm is that the conditions AC1, AC2(1), AC2(2) and AC3 defined in Section 2 can be mapped to computing sub- and superset relationships between good and bad execution traces. In the following we also use the terms sub-execution and super-execution to refer to sub- or superset relationships between execution traces. We define a number of execution trace comparison operators as follows.

Definition 12. *Execution Trace Comparison Operators. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* σ_1 *and* σ_2 *execution traces of* T.

$=_{\cdot}$: $\sigma_1 =_{\cdot} \sigma_2$ *iff* $\forall a \in \mathcal{A} \,.\, \sigma_1 \vDash_e a \equiv \sigma_2 \vDash_e a$.
$\doteq$: $\sigma_1 \doteq \sigma_2$ *iff* $\forall a_1, a_2 \in \mathcal{A} \,.\, \sigma_1 \vDash_e a_1 \wedge a_2 \equiv \sigma_2 \vDash_e a_1 \wedge a_2$.
$\subseteq_{\cdot}$: $\sigma_1 \subseteq_{\cdot} \sigma_2$ *iff* $\forall a \in \mathcal{A} \,.\, \sigma_1 \vDash_e a \Rightarrow \sigma_2 \vDash_e a$.
$\subset_{\cdot}$: $\sigma_1 \subset_{\cdot} \sigma_2$ *iff* $\sigma_1 \subseteq_{\cdot} \sigma_2$ *and not* $\sigma_1 =_{\cdot} \sigma_2$.
$\dot{\subseteq}$: $\sigma_1 \dot{\subseteq} \sigma_2$ *iff* $\forall a_1, a_2 \in \mathcal{A} \,.\, \sigma_1 \vDash_e a_1 \wedge a_2 \Rightarrow \sigma_2 \vDash_e a_1 \wedge a_2$.
$\dot{\subset}$: $\sigma_1 \dot{\subset} \sigma_2$ *iff* $\sigma_1 \dot{\subseteq} \sigma_2$ *and not* $\sigma_1 \doteq \sigma_2$.

In the following let φ a safety property specification given in LTL, $\sigma, \sigma', \sigma'', \sigma'''$ execution traces and $\psi_\sigma, \psi_{\sigma'}, \psi_{\sigma''}, \psi_{\sigma'''}$ the event order logic formulas representing these execution traces, respectively.

Theorem 1. *AC1 is fulfilled for all* ψ_σ *where* $\sigma \in \Sigma_B$.

Proof. For each $\sigma \in \Sigma_B$ we can partition the set $\mathcal{A}$ of event variables into the sets Z and W such that Z consists of the variables of the events that occur on σ and ψ_σ consists of the variables in Z. Consequently, $\sigma \vDash_e \psi_\sigma$ and $\sigma \nvDash_l \varphi$ because σ is a bad execution. Therefore, AC1 is fulfilled for all ψ_σ where $\sigma \in \Sigma_B$. □

Theorem 2. *AC2(1) holds for* ψ_σ *if there is an execution* $\sigma' \in \Sigma_G$ *with* $\sigma' \subset_{\cdot} \sigma$.

Proof. To show AC2(1) for a execution σ we need to show that there exists an execution σ' for which $\sigma' \nvDash_e \psi_\sigma \wedge (val_\sigma(Z) \neq val_{\sigma'}(Z) \vee val_\sigma(W) \neq val_{\sigma'}(W))$ and $\sigma' \vDash_l \varphi$ holds. For each $\sigma' \in \Sigma_G$ with $\sigma' \subset_{\cdot} \sigma$ there is at least one event on σ that does not occur on σ'. Because that missing event is part of ψ_σ and Z it follows $\sigma' \nvDash_e \psi_\sigma$ and $(val_\sigma(Z) \neq val_{\sigma'}(Z) \vee val_\sigma(W) \neq val_{\sigma'}(W))$ follows, since the value of the event variable representing the missing event assigned by $val_\sigma(Z)$ is *true* and the value assigned by $val_{\sigma'}(Z)$ is *false*. Therefore, we can show AC2(1) for ψ_σ by finding an execution $\sigma' \in \Sigma_G$ for which $\sigma' \subset_{\cdot} \sigma$ holds. □

Theorem 3. *AC2(2) holds for* ψ_σ *if there is no execution* $\sigma'' \in \Sigma_G$ *with* $\sigma \dot{\subset} \sigma''$.

Proof. AC2(2) requires that $\forall\sigma''$ with $\sigma'' \vDash_e \psi_\sigma \wedge (val_\sigma(Z) = val_{\sigma''}(Z) \wedge val_\sigma(W) \neq val_{\sigma''}(W))$ it holds that $\sigma'' \nvDash_l \varphi$ for all subsets of W. Suppose there exists a σ'' for which $\sigma \dot{\subset} \sigma''$ holds. For a σ'' to satisfy the condition $\sigma'' \vDash_e \psi \wedge val_\sigma(Z) = val_{\sigma''}(Z)$ all events that occur on σ have to occur in the same order on σ'', which is the case if $\sigma \dot{\subseteq} \sigma''$ holds. The set W contains the event variables of the events that did not occur on σ and $val_\sigma(W)$ assigns *false* to all event variables in W. For $val_{\sigma''}(W)$ to be different from $val_\sigma(W)$ there has to be at least one event variable that is set to *true* by $val_{\sigma''}(W)$. This is only the case if an event that does not occur on σ occurs on σ''. Consequently, σ'' consists of all events that did occur on σ and at least one event that did not occur on σ, which is true if $\sigma \dot{\subset} \sigma''$ holds. $\sigma'' \nvDash_l \varphi$ holds if $\sigma'' \in \Sigma_B$ and is false if $\sigma'' \in \Sigma_G$. Hence, AC2(2) holds for σ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{\subset} \sigma''$ holds. □

Theorem 4. *If AC1 and AC2(1) hold for ψ_σ and ψ_σ is modified according to Def. 10 in order to fulfill AC2(2), then AC1 and AC2(1) hold for the modified ψ_σ.*

Proof. The modification defined in Def. 10 prohibits the occurrence of events that did not occur on σ but occur on σ'' by adding their corresponding negated event variables to ψ_σ. Since the prohibited events did not occur on σ, the modified ψ_σ holds for σ and AC1 holds. AC2(1) holds for the modified ψ_σ because for AC2(1) to hold in the first place there has to be an execution $\sigma' \in \Sigma_G$ with $\sigma' \subset \sigma$. For the modification of ψ_σ to be necessary an execution $\sigma'' \in \Sigma_G$ with $\sigma \dot{\subset} \sigma''$ has to exist. If $\sigma \dot{\subset} \sigma''$ holds, $\sigma \subset \sigma''$ holds and $\sigma' \subset \sigma''$ holds as well. Consequently, AC2(1) holds for the modified ψ_σ. □

Theorem 5. *AC(3) holds for ψ_σ if there does not exists an execution $\sigma''' \in \Sigma_B$ for which $\sigma''' \subset \sigma$ holds.*

Proof. In AC(3) we have to show that no subset of the event order logic formula ψ satisfies AC1, AC2(1) and AC2(2). Suppose there exists a $\sigma''' \in \Sigma_B$ with $\sigma''' \subset \sigma$. We can partition $\mathcal{A}$ in $Z_{\sigma'''}$ and $W_{\sigma'''}$ such that $Z_{\sigma'''}$ consists of the variables of the events that occur on σ''' and $\psi_{\sigma'''}$ consists of the variables in $Z_{\sigma'''}$. For σ we partition $\mathcal{A}$ in Z_σ and W_σ such that Z_σ consists of the variables of the events that occur on σ and ψ_σ consists of the variables in Z_σ. Consequently, $Z_{\sigma'''} \subset Z_\sigma$ and $\psi_{\sigma'''} \subset \psi_\sigma$. If $\psi_{\sigma'''}$ satisfies AC1, AC2(1), AC2(2), then AC3 would be violated. If we can not find a $\sigma'''$with $\sigma''' \subset \sigma$, then no subset of ψ_σ satisfies AC1, AC2(1) and AC2(2), and consequently AC3 holds. □

We use these theorems in order to devise an algorithm and a corresponding data-structure called subset graph for on-the-fly causality checking. The pseudo-code for the proposed algorithms can be found in [9].

3.2 Subset Graph Data-Structure

In order to store the execution traces we have devised a data-structure called subset graph. This data-structure enables us to make causality decisions on-the-fly which means that we can decide whether an execution trace is causal as soon

as we add it to the subset graph. The subset graph is structured into levels where each level corresponds to the length of the execution traces on that level. Each node represents exactly one execution trace. Figure 1 shows a part of the subset graph for the railroad crossing example. The execution traces on adjoining levels are connected by edges indicating subset relationships between the respective execution traces. In order to improve readability the edges between executions on the same level are not displayed in the figure. The nodes representing the

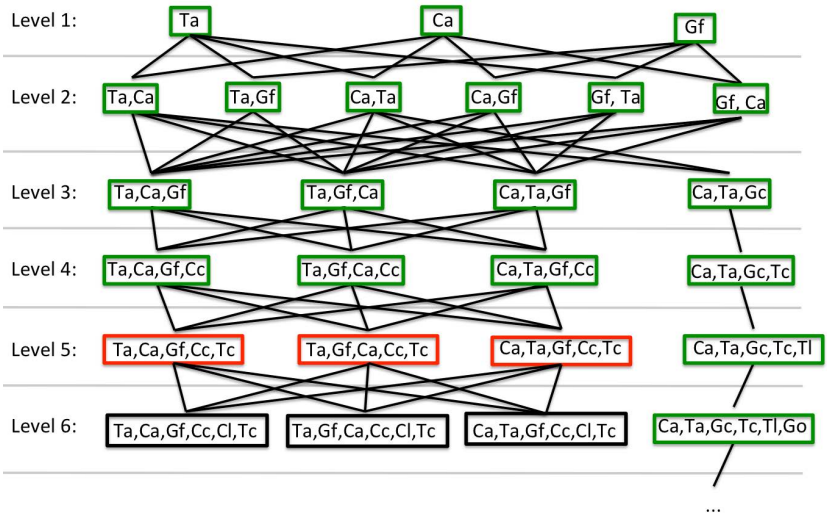


Fig. 1. Subset-graph of the railroad crossing example

execution traces are colored in green, red, black or orange in order to indicate their potential causality relation according to the following rules:

- Green: a node is colored *green* if it represents a good execution trace and all nodes on the level below that are connected with it are also colored green. An example of such a trace is "Ca,Ta,Gc,Tc,Tl" in the railroad crossing example. Green traces can not be causal because they are good traces. The green traces can be prefixes of either bad or good execution traces.
- Red: a node is colored *red* if it represents a bad execution trace and all nodes on the level below that are connected with it are colored green. Red nodes correspond to the shortest bad traces found at any point of the state-space exploration. They are considered to be causal. As an example consider the trace "Ta,Ca,Gf,Cc,Tc" in the railroad crossing example.
- Black: a node is colored *black* if it represents a good execution trace, but at least one node on the level below that it is connected with is colored red. Black traces cannot be causal themselves, since they are good traces, but since a sub-trace of them with one less element is a minimal bad trace, the transition in the subset graph from red to black identifies an event that turns

a bad execution into a good one. We hence take advantage of black traces when checking condition AC2(2). As an example for a black node consider the the trace "Ta,Ca,Gf,Cc,Cl,Tc" of the railroad crossing example, which is connected with the red execution "Ta,Ca,Gf,Cc,Tc" on the level below, the introduced "Cl" event turns the bad execution into a good one.

– Orange: A node is colored *orange* if it represents a bad execution trace and at least one node on the level below that is connected to the orange node is colored red. If a trace is colored orange, there exists a shorter red trace on a level below and hence a orange trace does not fulfill the minimality constraint AC3 for being causal. An example for an orange colored trace is the trace "Ca,Ta,Gc,Tc,Tl,Go,Ta,Gf,Cc,Tc" which, due to space restrictions, is not depicted in Figure 1. The trace "Ca,Ta,Gf,Cc,Tc" is a shorter red trace and a subset of the trace "Ca,Ta,Gc,Tc,Tl,Go,Ta,Gf,Cc,Tc", hence the trace does not fulfill the minimality constraint.

3.3 Causality Checking

The causality checking that we propose is embedded into both of the standard state-space exploration algorithms used in explicit state model checking, namely depth-first and breadth-first search. Whenever a bad or a good execution is found by the search algorithm it is added to the subset graph. After adding a trace the algorithm first checks whether there are executions at the same level that consist of the same events but in a different order. If we find such an execution, then all subset relationships of the execution already in the subset graph hold also for the newly added execution. For instance in our example all subset relationships that hold for the execution "Ta,Ca,Gf,Cc,Tc" also hold for the execution "Ta,Gf,Ca,Cc,Tc". If we don't find such a trace on the same level, we have to check the subset relationships with the execution traces on the level below (level-1) and, if depth-first search is used, on the level above (level+1) as well. It is not necessary to check the subset relationships on the level above (level+1) if breadth-first search is used, because breadth-first search adds the traces by increasing length.

Once all subset relationships are established, the nodes representing the executions are colored according to the above described coloring rules. If a trace is colored red, we additionally need to check whether we have already found a shorter red trace which is a sub-set of the new red-trace. If such a shorter red trace is found, the current trace is colored orange. In our example the execution traces *Ta,Ca,Gf,Cc,Tc* and *Ta,Gf,Ca,Cc,Tc* and *Ca,Ta,Gf,Cc,Tc* are colored red and hence considered to be causal.

The following theorems show that for an execution σ that is colored red, ψ_σ is a candidate for being causal and fulfills AC1, AC2(1) and AC3.

Theorem 6. *AC1 is fulfilled for ψ_σ of each execution trace σ that is colored red.*

Proof. By definition an execution trace is only colored red if it is a bad trace and according to Theorem 1 AC1 is fulfilled for all $\sigma \in \Sigma_B$. □

Theorem 7. *AC2(1) is fulfilled for ψ_σ of each execution trace σ that is colored red.*

Proof. According to Theorem 2 we can show AC2(1) by finding an execution $\sigma' \in \Sigma_G$ for which $\sigma' \sqsubset \sigma$ holds. For an execution σ to be colored red, all sub execution traces on the level below have to be colored green. Consequently, for each execution σ' for which $\sigma' \sqsubset \sigma$ holds also $\sigma' \in \Sigma_G$ holds because it is colored green and hence needs to be a good trace. Therefore, AC2(1) is fulfilled according to Theorem 2. □

Theorem 8. *If breadth-first search is used, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red. If depth-first search is used, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red as soon as the state-space exploration has terminated.*

Proof. According to Theorem 5, ψ fulfills AC3 if there does not exists a trace $\sigma''' \in \Sigma_B$ for which $\sigma''' \sqsubset \sigma$ holds. This is due to the fact that by definition an execution trace is only colored red if all its subsets are colored green, which means there is no bad sub-execution σ''' of σ. If breadth-first search is used the shortest paths are added first, hence all sub-executions are known at the time where σ is inserted and colored. Consequently, if breadth-first search is used, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red. If depth-first search is used it is possible that new sub-executions are found as long as the state-space exploration is not complete. As a result, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red as soon as the state-space exploration with depth-first search has terminated. □

Once the state space search is completed we have to perform the tests for AC2(2) and OC1 for all red execution traces.

According to Theorem 3, AC2(2) holds for ψ_σ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{\sqsubset} \sigma''$ holds. If such a σ'' exists, it is a black superset of σ because $\sigma \sqsubset \sigma''$ holds for each black superset of σ. σ'' is only colored black if it is a good trace. Consequently, we need to check for each black superset σ'' of σ whether $\sigma \dot{\sqsubset} \sigma''$ holds. If there is no σ'' for which $\sigma \dot{\sqsubset} \sigma''$ holds, then ψ_σ fulfills AC2(2). If $\sigma \dot{\sqsubset} \sigma''$ holds for a black superset, then we need to modify ψ_σ as specified by Definition 10. Hence, we have shown that AC1, AC2(1), AC2(2) and AC3 are fulfilled for ψ_σ of each red execution σ and, consequently, that ψ_σ is causal for the property violation.

Notice that the AC2(2) test is needed in order to detect whether the non-occurrence of an event is causal. It is necessary to store all traces that are colored black only to test AC2(2). We have added a runtime switch in the implementation of the causality checking method that allows the user to turn the AC2(2) test off in order to save memory at the expense of not being able to take the possible causality of the non-occurrence of an event into account. If the AC2(2) test is fulfilled by ψ_σ, then the OC1 test is performed. Due to the structure of the subset graph, it is sufficient to check for each red execution trace whether there exists a red execution trace on the same level for which the unordered $\subseteq$ relationship

holds. For all those execution traces, we check for each pair of events whether they appear on all execution traces in the same order or not. If a pair of events does not occur in the same order, then the order of this pair is marked as having no influence on causality.

Causality Checking with Breadth-First Search (BFS). When using breadth-first search, the execution trace leading from the initial state to a property violating state can be generated by iterating backwards through the predecessor links until an initial state is reached. Whenever a bad or a good execution is found, it is added to the subset graph. If BFS encounters a state that is already in the state-space and hence all successors of this state have already been explored, the successors are not explored for a second time. Since BFS explores the state-space following an exploration order that leads to a monotonically increasing length of the execution traces, this new execution trace reaching the state either has the same length as the already known execution trace containing the same state, or the new execution is longer than the already known execution trace. If the new execution trace has the same length, the events on the trace have an order that is different from the one in the already known execution trace. Hence the new execution trace needs to be added to the subset graph since a later OC1 test needs to be performed on it.

Causality Checking with Depth-First Search (DFS). We adapted the depth-first search algorithm to add an execution trace to the subset graph data structure whenever either a bad state is reached or a good execution trace has been found. If depth-first search is used it is sufficient to print the search stack in order to retrieve the execution trace. Similarly to BFS, if DFS encounters a duplicate, which is a state that is already in the state-space, and hence all successors of the duplicate have already been explored, the successors are not explored a second time. It is possible that this new trace to the duplicate is shorter or has a different event order than the already known execution traces that contain the duplicate. Hence we store this new execution trace on a match list in the subset graph and generate all execution traces starting from the duplicate state with the new trace as a prefix.

Complexity. [10] contains a careful analysis of the complexity of computing causality in the SEM. Most notable is the result that even for an SEM with only binary variables, in the general case computing causal relationships between variables is NP-complete. Results in [11] show that causality can be computed in polynomial time if the causal graph over the events forms a directed causal tree. A directed causal tree consists of directed paths, where the nodes represent events, and the edges represent the causality relationships and the root node represents the hazard or effect. Each bad execution trace in the counterexample is a directed path containing the variables representing the events leading to the hazard or effect. Consequently, a set of counterexamples resembles a directed causal tree and our algorithm can compute the causal process in polynomial time. A more comprehensive discussion of the complexity of our approach can be found in [9].

4 Case Studies

In order to evaluate the proposed approach, we have implemented our causality checking algorithms within the SpinJa toolset [12], a Java re-implementation of the explicit state model checker Spin [5]. Our SpinCause tool[1] computes the causality relationships for a Promela model and a given LTL property. In order to compute all interleavings and all executions partial-order reduction was disabled during the state-space exploration. The Promela models used for the case studies have been created manually, in practical usage scenarios the Promela models can also be automatically synthesized from higher-level design models, as for instance by the QuantUM tool [13]. The following experiments were performed on a PC with an Intel Xeon Processor (3.60 Ghz) and 144 GBs of RAM.

4.1 Railway Crossing

The Promela model of the railway crossing that we constructed as a running example for the purpose of this paper comprises 133 states and 237 transitions. A total of 47 bad execution traces are found. The causality checking algorithm identified two event order logic formulas describing the causal factors for a train and a car being on the crossing at the same time.

- First, if the gate fails at some point of the execution and a train (Ta) and a car (Ca) are approaching this results in a hazardous situation if the car is on the crossing (Cc) and does not leave the crossing (Cl) before the train (Tc) enters the crossing $(\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \owedge \mathrm{Cc})) \owedge_{<} \neg \mathrm{Cl} \owedge_{>} \mathrm{Tc}))$.
- Second, if a train (Ta) and a car (Ca) are approaching but the gate closes (Gc) when the car (Cc) is already on the railway crossing and is not able to leave (Cl) before the gate is closing and the train is crossing (Tc), this also corresponds to a hazardous situation $((\mathrm{Ta} \wedge (\mathrm{Ca} \owedge \mathrm{Cc})) \owedge_{<} \neg \mathrm{Cl} \owedge_{>} (\mathrm{Gc} \wedge \mathrm{Tc}))$.

4.2 Airbag Control Unit

The industrial size model of an airbag system that we use in this case study is taken from [14]. The architecture of this system was provided by our industrial partner TRW Automotive GmbH. The architecture of this system consists of two acceleration sensors, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag. Although airbags save lifes in crash situations, they may cause fatal accidents if they are inadvertently deployed. This is because the driver may lose control of the car when this deployment occurs. It is a pivotal safety requirement that an airbag is never deployed if there is no crash situation. We are interested in computing the causal events for the hazard corresponding to an inadvertent ignition of the airbag. The Promela model of the airbag system consists of 155,464 states and 697,081 transitions.

Figure 2 shows the fault tree generated by the SpinCause tool. All execution traces that are colored red are part of the fault tree representation. The fault

[1] http://se.uni-konstanz.de/research1/tools/spincause

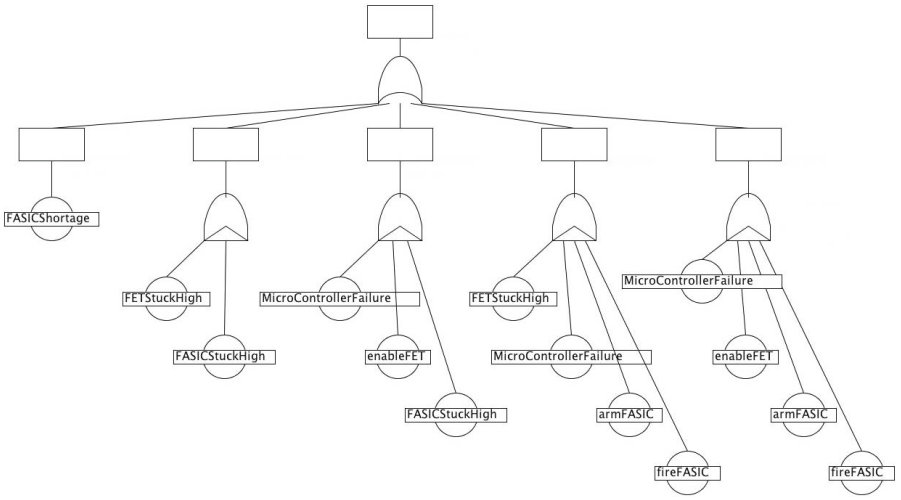


Fig. 2. Fault tree of the airbag system

trees generated by our approach all have a normal form, that is they start with an *intermediate* gate representing the top level event, that is connected to an *OR* gate. The execution traces that are colored red are represented by Priority-AND (PAND) gates if the order of some events is causal and by AND gates if the order is not causal. The events of the execution traces are connected to the corresponding AND or PAND gates, respectively. Since fault trees are not sufficiently expressive to completely represent an event oder logic formula, we display for each PAND gate the event order logic formula constraining the order of the events connected to the PAND-gate (omitted in Figure 2 for better readability).

While there are a total of 20,300 bad execution traces, the fault tree comprises only 5 paths. Obviously, a manual analysis of this large number of traces in order to determine causal factors would be impossible. It is easy to see in the fault tree which basic events cause an inadvertent deployment of the airbag. For instance, there is only one single fault that can lead to an inadvertent deployment, namely *FASICShortage*, which is represented by the event order logic formula FASICShortage. It is also obvious that the combination of the basic events *FETStuckHigh* and *FASICStuckHigh* only leads to an inadvertent deployment of the airbag if the basic event *FETStuckHigh* occurs prior to the basic event *FASICStuckHigh*, which is represented by the event order logic formula FETStuckHigh $\wedge$ FASICStuckHigh. The basic event *MicroControllerFailure* can lead to an inadvertent deployment if it is followed by the following sequence of basic events: *enableFET*, *armFASIC*, and *fireFASIC*. This sequence is represented by the event order logic formula MicroControllerFailure $\wedge$ enableFET $\wedge$ armFASIC $\wedge$ fireFASIC. If the basic event *FETStuckHigh* occurs prior to the *MicroControllerFailure* the sequence in which *armFASIC* and *fireFASIC* occur after the *MicroControllerFailure* event suffices to lead to the top level event.

This sequence is represented by the event order logic formula FETStuckHigh ∧ MicroControllerFailure ∧ armFASIC ∧ fireFASIC. If the basic event *FASICStuck-High* occurs after *MicroControllerFailure* and *enableFET* this also leads to a sequence leading to an inadvertent deployment. It is represented by the event order logic formula MicroControllerFailure ∧ enableFET ∧ FASICStuckHigh.

The case study shows that the fault tree is a compact and concise visualization of the counterexample which allows for an easy identification of the basic events that cause the inadvertent deployment of the airbag. If the order of the events is important for the events causing the effect, this can be seen in the fault tree by the *PAND* gate and the corresponding EOL formula. In the counterexamples computed by SpinJa one would have to manually compare the order of the events in all execution traces.

4.3 Discussion

Table 1 shows the memory and run time consumption of the on-the-fly causality checking approach presented in this paper for both case studies and the memory and run time consumption of the in off-line approach presented in [4], where all execution traces are stored on disk during model checking (Run. MC., Mem. MC) and the causality checking is performed as a post-processing step (Run. Caus., Mem. Caus.), for the airbag case study. The following trends can be identified:

Table 1. This table shows the experiment results with the on-the-fly approach for the railway crossing and airbag case studies. Run. MC and Mem. MC show the runtime and memory consumption for model checking only. Run. CC1 and Mem. CC1 show the runtime and memory needed to perform model checking and causality checking with the AC2(2) test disabled and Run. CC2 and Mem. CC2 with the AC2(2) test enabled. Additionally, the experiment results for off-line causality checking of the airbag case study are given.

	On-the-fly Approach						Off-line Approach			
	Run time (sec.)			Memory (MB)			Run time		Memory (MB)	
	MC	CC 1	CC2	MC	CC1	CC 2	MC	Caus.	MC	Caus.
Airbag										
DFS	0.98	338.17	597.57	25.08	15,711.20	27,687.50	871.14	945.68	1,478.34	28,563.47
BFS	0.96	148.52	195.05	25.74	1,597.54	3,523.04	486.01	512.3	1,331.29	13,860.10
Railway										
DFS	0.01	0.29	0.31	16.40	20.38	21.68	-	-	-	-
BFS	0.01	0.12	0.13	16.24	16.70	17.45	-	-	-	-

– If no causality checking is done, DFS and BFS have approximately the same runtime and memory consumption. The causality checking adds a run-time and memory penalty, but the experiments show that causality checking is applicable to industrial size Promela models. In addition causality checking provides valuable insight as to why the hazard occurred, which is very tedious

or even impossible to determine if standard model checking and manual counterexample analysis is used.
- When performing causality checking, BFS outperforms DFS in terms of both runtime and memory consumption. BFS outperforms DFS because if BFS is used, we can safely rely on the assumption that when a bad trace is found all shorter bad traces already have been found. This assumption assures that the minimality condition holds for each bad trace which was found using BFS and colored red by the causality checking algorithm. If DFS is used, no assumptions on the length of the bad trace can be made. The main reason why the assumption on the bad trace length is important and has such a high impact on the memory consumption when using DFS compared to BFS is that all good traces which are supersets of a red trace have to be taken into account for the AC2(2) test. When BFS is used only the traces which are supersets of red traces need to be stored, whereas when DFS is used all good traces need to be stored. Because the good traces are needed in case a shorter red trace is found later in the search for which we need the good super-traces for the AC2(2) test.
- The on-the-fly approach proposed in this paper outperforms the off-line approach both in terms of runtime and memory consumption. The main reason for this observation is that when using the on-the-fly approach only the execution traces needed for causality checking, namely the red and black execution traces, need to be stored, whereas all execution traces have to be stored for the off-line approach.

5 Related Work

The application of counterfactual reasoning to software debugging has been proposed by Zeller in [15]. However, [15] does not support complex logical relationships as causes and is mainly applicable to sequential software programs, whereas our approach is also applicable to concurrent software and hardware systems. Work documented in [16] uses the Halpern and Pearl approach to explain counterexamples in CTL model checking by determining causality. However, this approach considers only single counterexamples. Furthermore, it focuses on the causality of variable value-changes for the violation of CTL sub-formulas, whereas our approach identifies the events that lead to the variable value-changes. Consider the railway crossing example in which the CTL formula consists of the two boolean variables train_on_crossing and car_on_crossing. Obviously, both variables changing to true is causal for a crash. Consequently the approach from [16] will indicate the variable value-change of train_on_crossing and car_on_crossing from false to true as being causal. But this obvious answer does not give any insight on why the train and the car are on the crossing at the same time. In [17] a formal framework for reasoning about contract violations is presented. In order to derive causality the notion of precedence established by Lamport clocks [18] is used. While this captures a partial order of the observed contract violations it is not clear to what extent this order information also

expresses causality. Work described in [19] establishes causality based on counterfactual reasoning by computing distance metrics between execution traces. The delta between the counterexample and the most similar good execution is identified as causal for the bad behavior. For all the above mentioned approaches it is necessary to compute the counterexamples prior to the causality analysis whereas our approach works on-the-fly. To the best of our knowledge we are not aware of any other causality checking algorithm that can be integrated with explicit state-space exploration algorithms and which works on-the-fly. As an alternative to the event order logic that we defined we also investigated the usage of the interval logics [20] and [21]. We felt that in light of the relatively simple ordering constraints that we need to describe those logics are overly expressive, and we hence decided to define our own tailored, relatively simple event order logic.

6 Conclusions

We have discussed how causality relationships can be established in system executions and have shown how the causality checks can be mapped to finding sub- and super-sets of execution traces. Furthermore we have proposed an approach for causality computation that works on-the-fly and can be integrated into explicit state-space model checking algorithms. We have evaluated our approach on two case studies, one of which is of industrial size. The experimental evaluation indicates that breadth-first search outperforms depth-first search in terms of memory and runtime, and that the on-line approach presented here outperforms the precurosy off-line approach. Furthermore, we have shown that causality checking is applicable to industrial size Promela models.

In future work we plan to give a soundness and completeness argument for causality checking and embed causality checking into a symbolic reasoning environment in order to avoid the explicit storing of traces. In addition we plan to combine our work on causality checking for probabilistic models with the approach presented here.

Acknowledgment. We wish to thank Stefan Heindorf for a careful review of an earlier version of this work.

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking, 3rd edn. The MIT Press (2001)
2. Lewis, D.: Counterfactuals. Wiley-Blackwell (2001)
3. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural-model approach. Part I: Causes. The British Journal for the Phil. of Science (2005)
4. Kuntz, M., Leitner-Fischer, F., Leue, S.: From Probabilistic Counterexamples via Causality to Fault Trees. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 71–84. Springer, Heidelberg (2011)

5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addision–Wesley (2003)
6. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
7. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc. (1992)
8. Collins, J. (ed.): Causation and Counterfactuals. MIT Press (2004)
9. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. Chair for Software Engineering, University of Konstanz, Technical Report soft-12-02 (2012), http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-12-02.pdf
10. Eiter, T., Lukasiewicz, T.: Complexity results for structure-based causality. Artificial Intelligence (2002)
11. Eiter, T., Lukasiewicz, T.: Causes and explanations in the structural-model approach: Tractable cases. Artificial Intelligence (2006)
12. de Jonge, M., Ruys, T.C.: The SpinJa Model Checker. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 124–128. Springer, Heidelberg (2010)
13. Leitner-Fischer, F., Leue, S.: QuantUM: Quantitative safety analysis of UML models. In: Proc. of the 9th Workshop on Quantitative Aspects of Programming Languages, QAPL 2011 (2011)
14. Aljazzar, H., Fischer, M., Grunske, L., Kuntz, M., Leitner-Fischer, F., Leue, S.: Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In: Proc. of QEST 2009. IEEE Computer Society (2009)
15. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Elsevier (2009)
16. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining Counterexamples Using Causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009)
17. Gössler, G., Le Métayer, D., Raclet, J.-B.: Causality Analysis in Contract Violation. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 270–284. Springer, Heidelberg (2010)
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21, 558–565 (1978)
19. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. International Journal on Software Tools for Technology Transfer (STTT) 8(3) (2006)
20. Schwartz, R.L., Melliar-Smith, P.M., Vogt, F.H.: An interval logic for higher-level temporal reasoning. In: Proc. of the 2nd Annual ACM Symposium on Principles of Distributed Computing. ACM (1983)
21. Dillon, L., Kutty, G., Moser, L., Melliar-Smith, P., Ramakrishna, Y.: A graphical interval logic for specifying concurrent systems. ACM Transactions on Software Engineering and Methodology (TOSEM) 3(2), 131–165 (1994)

ClaburéDB: Classified Bug-Reports Database
Tool for Developers of Program Analysis Tools

Jiri Slaby, Jan Strejček, and Marek Trtík

Faculty of Informatics, Masaryk University
Botanická 68a, 60200 Brno, Czech Republic
{slaby,strejcek,trtik}@fi.muni.cz

Abstract. We present a database that can serve as a tool for tuning and evaluation of miscellaneous program analysis tools. The database contains bug-reports produced by various tools applied to various source codes. The bug-reports are classified as either real errors or false positives. The database currently contains more than 800 bug-reports detected in the Linux kernel 2.6.28. Support of other software projects written in various programming languages is planned. The database can be downloaded and manipulated by SQL queries, or accessed via a web frontend.

1 Introduction

Many successful bug-finding tools based on various program analysis methods appeared during the last ten years. None of them is perfect. Each tool either reports both real errors and false positives, or it discovers only a part of real errors. To improve or evaluate such a tool, one needs to run the tool on some source codes and then analyze the obtained bug-reports[1], i.e. classify them as false positives or real errors, and find errors in the sources that were missed by the tool. This work is usually tedious and time consuming, especially when one tunes or studies performance of a tool for real software projects. The tedious work can be avoided if suitable benchmarks, i.e. programs with information about their errors, are available.

There exist benchmark suites consisting of small synthetic programs [1,2,4,6,7] and those consisting of real-world programs [2,3,5]. In benchmark suites [1,4,6], relevant program locations in small synthetic programs are explicitly marked as either erroneous or safe. The benchmark suite [2] marks only known real errors. The situation is different for benchmarks with real-world programs: [5] contains marked test cases triggering/non-triggering errors, while [3] provides bug-reports (both real errors and false positives mixed together) produced by FindBugs and sorted according to priority levels assigned to the reports by the tool. The benchmark suites discussed so far consider different error types and

[1] We deliberately use term "bug" in the paper. It stems from the need of the database to comprehend for example coding style violations. In those cases, commonly used terms like "failure" or "fault" fail to apply.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 268–274, 2013.

provide their own error taxonomies with exception of suites [1,2], where errors types are linked to *Common Weakness Enumeration* (CWE) [10].

As far as we know, there is no benchmark suite containing big real-life projects with a significant list of uniformly described bug-reports classified as real errors or false positives. Our benchmark suite ClaburеDB should fill this gap.

ClaburеDB currently contains only a single project, namely the Linux kernel 2.6.28. We have collected about 850 bug-reports of 11 error types produced by several bug-detection tools run on the kernel. The reports have been manually classified as either real errors or false positives by skilled programmers with a help of Linux kernel developers. In fact, it would be sufficient to store only real errors assuming that we know all of them (and thus we can assume that all other bug-reports are false positives). As this assumption is completely unrealistic for a real-life project, we store both real errors and false positives.

The database is still developing in several directions: we plan to add more bug-reports for the Linux kernel, to support other software projects, and to augment the web interface to allow other users to add and maintain the database content.

The database can be downloaded in the SQLite 3 format for local use under the *Open Database License v1.0* [11] or accessed via a web interface at:

`http://claburedb.fi.muni.cz/`

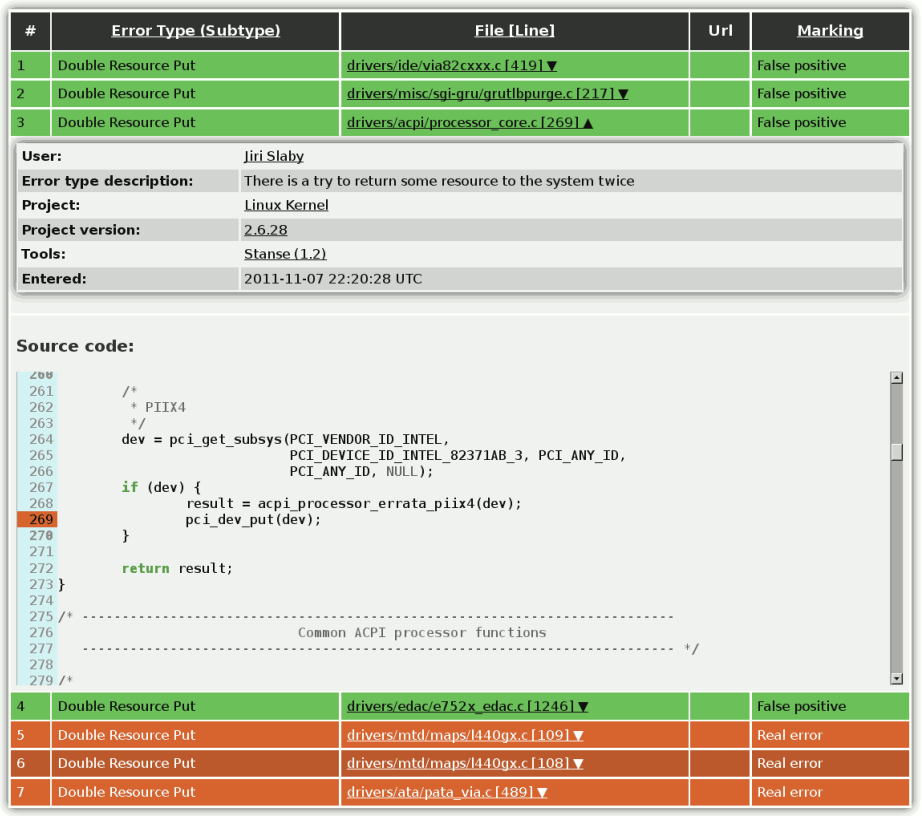
The paper is organized as follows. The following section introduces the basic structure of the database and our web interface. Section 3 describes the current content of the database including considered kinds of errors and an overview of collected bug-reports and their sources. Section 4 suggests possible use of the database for evaluation of a program analysis tool. Finally, the last section presents our future plans with ClaburеDB.

2 Database Structure

The database is designed to accommodate various kinds of errors from diverse projects and project versions. As projects can be written in arbitrary programming languages, can contain very specific kinds of errors, can be maintained by different teams, and can be interesting for distinct user groups, we decided to have each project in a separate sub-database. A sub-database comprises information about considered error types, bug-reports, users, tools, and relations between them. There are two main tables in each sub-database:

error_type. This table keeps a specification of considered kinds of errors. We will outline some of possible types later in Section 3.2. The table contains a name of the type, short description and a reference CWE number if exists (see later).

error. Each line in this table corresponds to one bug-report. It is specified by the error type, location (usually file and line), URL for reference (to provide more information), classification (false positive, real error, unclassified), confirmation (an argument supporting the validity of the bug-report classification, e.g. a commit ID of the corresponding bug-fix), user who inserted the entry, and timestamp of the moment of insertion.



#	Error Type (Subtype)	File [Line]	Url	Marking
1	Double Resource Put	drivers/ide/via82cxxx.c [419] ▼		False positive
2	Double Resource Put	drivers/misc/sgi-gru/grutlbpurge.c [217] ▼		False positive
3	Double Resource Put	drivers/acpi/processor_core.c [269] ▲		False positive

User:	Jiri Slaby
Error type description:	There is a try to return some resource to the system twice
Project:	Linux Kernel
Project version:	2.6.28
Tools:	Stanse (1.2)
Entered:	2011-11-07 22:20:28 UTC

Source code:

```
260
261         /*
262          * PIIX4
263          */
264         dev = pci_get_subsys(PCI_VENDOR_ID_INTEL,
265                              PCI_DEVICE_ID_INTEL_82371AB_3, PCI_ANY_ID,
266                              PCI_ANY_ID, NULL);
267         if (dev) {
268                 result = acpi_processor_errata_piix4(dev);
269                 pci_dev_put(dev);
270         }
271
272         return result;
273 }
274
275 /* --------------------------------------------------------------------------
276                               Common ACPI processor functions
277    -------------------------------------------------------------------------- */
278
279 /*
```

4	Double Resource Put	drivers/edac/e752x_edac.c [1246] ▼		False positive
5	Double Resource Put	drivers/mtd/maps/l440gx.c [109] ▼		Real error
6	Double Resource Put	drivers/mtd/maps/l440gx.c [108] ▼		Real error
7	Double Resource Put	drivers/ata/pata_via.c [489] ▼		Real error

Fig. 1. List of seven bug-reports with one record expanded showing detailed information about the bug-report

Figure 1 depicts a list of seven bug-reports of type *Double Resource Put* as presented in the ClaburedB's web interface at `http://claburedb.fi.muni.cz/`. Four of these reports are false positives (green), the other three are real errors (orange). One of the reports is expanded, so that we can see its details and a highlighted source code with a marked error line.

Recall that the database can be also downloaded in the SQLite 3 format for local use under the *Open Database License v1.0* [11].

3 Current Contents of the Database

ClaburedB currently contains only bug-reports produced for the Linux kernel 2.6.28. We have chosen Linux kernel for several reasons: it is a big (20484 files with nearly 9 millions LOC in total), real-life, self-contained, and open-source project with several public sources of information about errors. Moreover, the kernel contains many distinct parts (core, hardware drivers, filesystems,

networking etc.) that cover some standard application areas of program analysis tools.

3.1 Sources of Bug-Reports

We used three program analysis tools to gather bug-reports for the kernel: Clang 3 [9], Stanse 1.2 [8], and one commercial tool which wanted to stay in anonymity. Note that Clang does not natively detect locking errors in the Linux kernel. Hence, we slightly modified `experimental.unix.PthreadLock` checker just to understand kernel locking functions. The database also comprises bug-reports from kernel and Novell bug tracking systems and mailing lists. For that purpose we implemented our own web crawler. Finally, there are few bug-reports detected manually.

3.2 Linux Kernel Error Types

We have analyzed output of available tools applicable to the Linux kernel and we have decided to focus on the following 11 error types. Nine of these error types are present in *Common Weakness Enumeration* (CWE) [10] and we provide their CWE identification numbers. The two remaining error types are specific for the Linux kernel (they can be seen as a violation of the kernel coding policy) and thus they are not covered by CWE.

The list of considered error types follows. For each error type we explicitly describe the program location associated with an error of this type. This is to evade misinterpretation, because diverse tools can associate the same error with different program locations. For example, an error present at an outgoing edge of a function may be associated to the opening brace of the function, the closing brace, or to the corresponding `return` statement.

BUG/WARNING (CWE 617) Developers often inject *assert*s to their code, e.g. to ensure that their function is given a correct input. For example, a destroy function of an object `obj` can contain a line `assert(!obj->active)` to check that the object to be destroyed is inactive. An error occurs if a condition of some `assert` is violated.
Error location: the line with the violated assertion.

Division by Zero (CWE 369) The code contains a division instruction but the actual value of the divisor is zero.
Error location: the line with the division.

Double Lock (CWE 764) Some lock is locked by a thread twice in a row and it leads to an inconsistent lock state. Note that we ignore double locks of semaphores as this may be their intentional application.
Error location: the line with the second call of lock.

Double Unlock (CWE 765) Some lock is unlocked by a thread twice in a row and it leads to an inconsistent lock state. Again, we ignore double unlocks of semaphores.
Error location: the line with the second call of unlock.

Double Free (CWE 415) A freeing function is called twice on the same address while no reassignment to the passed pointer occurred between the two calls. Most allocators can detect this and usually kill the program.
Error location: the line of the invalid (second) free.

Memory Leak (CWE 401) Some code omits to free a memory which was allocated previously.
Error location: the line with the allocation statement.

Invalid Pointer Dereference (CWE 465) The code tries to access some memory, but the pointer used is invalid. It may become invalid in many ways. For example, the pointer may point to a released memory (known as *dangling pointer* or *use after free*) or it can be set to NULL (known as *NULL pointer dereference*) or uninitialized. Another source of the problem may be accessing an array out of bounds.
Error location: the line of the dereference.

Double Resource Put (CWE 763) The code requests one copy of a resource (e.g. a structure holding hardware status) from a system, but there is more than one attempt to put that resource back to the system. Like in this example:

```
struct pci_dev *pdev = pci_get_device(vendor, device, NULL); // get
if (pdev) {
  work_with_pdev(pdev)
  pci_dev_put(pdev);                // first put
}
pci_dev_put(pdev);                  // second (illegal) put of the same
```

Error location: the line of the second put.

Resource Leak (CWE 404) The code requests a resource from a system, but omits to return that back. For example, the error occurs in the previous example if we remove both `pci_dev_put` calls.
Error location: the line of the request/get.

Calling Function from Invalid Context (not in CWE, Linux kernel specific) Some function is called at an inappropriate place or within an invalid context. This includes calling functions like sleep or wait inside spin-lock critical sections or in interrupt handlers. This is considered to be an error because results of such a call are undefined: the system may become unresponsive or may crash for instance.
Error location: the line of the inappropriate call.

Leaving Function in Locked State (not in CWE, Linux kernel specific) This error type originates from the kernel requirements that a process has to release all locks before returning control to userspace. This error occurs if a function has an execution path where some lock is locked and left locked when leaving the function. It is considered to be an error (violation of the kernel coding policy) as such an execution may lead to a deadlock on the next invocation of any function wanting to take the same lock.
Error location: the line of the corresponding `return` statement or closing brace if there is no `return`.

Table 1. Reports in ClabureDB

Error Type	Real Err.	False Pos.	Unclassified	Total
BUG/WARNING	8	0	0	8
Division by Zero	2	0	0	2
Double Lock	16	95	4	115
Double Unlock	22	90	9	121
Double Free	0	1	0	1
Memory Leak	7	13	0	20
Invalid Pointer Dereference	17	17	0	34
NULL Pointer Dereference	17	14	0	31
Use After Free	0	3	0	3
Double Resource Put	3	4	0	7
Resource Leak	13	51	24	88
Calling function from invalid context	16	19	0	35
Leaving function in locked state	30	352	37	419
Overall Count	134	642	74	850

3.3 Bug-Reports in the Database

Currently the database comprises 850 reports: 134 real errors, 642 false positives, and 74 entries which are unclassified. Many of the unclassified entries were added even recently and are about to be classified soon. Table 1 depicts how each kind of bug-reports is represented in the database. As seen from the table, most of the bug-reports in the database are related to locking errors.

4 Intended Use of the Database

Typical use of ClabureDB is tuning and evaluation of a bug-finding program analysis tool. To evaluate such a tool, one chooses a project (or its part) from our benchmark suite and run the tool on these source codes. As the second step, the sub-database of classified bug-reports corresponding to the chosen project is downloaded from ClabureDB. The installation and local database usage is described in the ClabureDB documentation.

The bug-reports produced by the tool are compared to the downloaded data. This way, one immediately gets a classification (real error/false positive) of all the reports matched in the database and also information about missed real errors if there are any. It remains to manually classify the bug-reports unmatched with the database content. The numbers of produced false positives, detected and missed real errors can be further statistically processed according to methodologies of [2,3] which produce several metrics including accuracy and precision. The missed errors and false positives are valuable inputs for tuning the tool.

Finally, the user is kindly asked to insert newly discovered bug-reports with their classification into the database.

5 Conclusion and Future Plans

We have presented ClaburéDB: the database of classified bug-reports. It is intended as an open platform for sharing valuable benchmarks for developers of program analysis tools. The database already contains a large collection of uniformly described bug-reports produced by various tools on a large real-life project, namely the Linux kernel 2.6.28.

We plan to extend the database in several directions. Namely, we intend to collect more bug-reports and to support more error types in the Linux kernel project. For example, we plan to add error types connected to deadlock and livelock. Further, we plan to add other real-life projects. We also plan to extend the web interface to enable developers of program analysis tools to maintain and augment the database.

In general, the future of ClaburéDB depends on a feedback from the program analysis community. At this point, we would like to encourage you to contact us at `claburedb@fi.muni.cz` if you have any comments, suggestions (for example which projects should be added to the database), questions, or sources of bug-reports for the Linux kernel or other interesting projects. We would especially welcome people willing to participate on creating and maintaining the database content for another project.

Acknowledgements. All authors have been supported by The Czech Science Foundation (GAČR), grant No. P202/12/G061.

References

1. Chatzieleftheriou, G., Katsaros, P.: Test-driving static analysis tools in search of C code vulnerabilities. In: Proceedings of COMPSACW, pp. 96–103. IEEE Computer Society (2011)
2. Cifuentes, C., Hoermann, C., Keynes, N., Long, S., Li, L., Mealy, E., Mounteney, M., Scholz, B.: BegBunch – Benchmarking for C Bug Detection Tools. In: Proceedings of DEFECTS, pp. 16–20. ACM (2009)
3. Heckman, S., Williams, L.: On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: Proceedings of ESEM, pp. 41–50. ACM (2008)
4. Kratkiewicz, K.: Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In: Proceedings of BUGS (2005)
5. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on ESDDT (2005)
6. Newsham, T., Chess, B.: ABM: A prototype for benchmarking source code analyzers. In: Proceedings of SSATTM, pp. 52–59. NIST Special Publication (2005)
7. NIST. Samate reference dataset project, `http://samate.nist.gov/SRD/`
8. Obdržálek, J., Slabý, J., Trtík, M.: STANSE: Bug-Finding Framework for C Programs. In: Kotásek, Z., Bouda, J., Černá, I., Sekanina, L., Vojnar, T., Antoš, D. (eds.) MEMICS 2011. LNCS, vol. 7119, pp. 167–178. Springer, Heidelberg (2012)
9. Clang: a C language family frontend for LLVM, `http://clang.llvm.org/`
10. Common Weakness Enumeration (CWE), `http://cwe.mitre.org/`
11. Open Database License 1.0, `http://opendatacommons.org/licenses/odbl/1.0/`

Tool Integration with the Evidential Tool Bus*

Simon Cruanes[1], Gregoire Hamon[2], Sam Owre[2], and Natarajan Shankar[2]

[1] Ecole Polytechnique, Palaiseau, France
[2] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
simon.cruanes.2007@polytechnique.org,
{Hamon,Owre,Shankar}@csl.sri.com

Abstract. Formal and semi-formal tools are now being used in large projects both for development and certification. A typical project integrates many diverse tools such as static analyzers, model checkers, test generators, and constraint solvers. These tools are usually integrated in an *ad hoc* manner. There is, however, a need for a tool integration framework that can be used to systematically create workflows, to generate claims along with supporting evidence, and to maintain the claims and evidence as the inputs change. We present the Evidential Tool Bus (ETB) as a tool integration framework for constructing claims supported by evidence. ETB employs a variant of Datalog as a metalanguage for representing claims, rules, and evidence, and as a scripting language for capturing distributed workflows. ETB can be used to develop assurance cases for certifying complex systems that are developed and assured using a range of tools. We describe the design and prototype implementation of the ETB architecture, and present examples of formal verification workflows defined using ETB.

Keywords: Certification, Formal techniques, Hybrid techniques, Tool integration, Workflow.

1 Introduction

Formal techniques such as model checking, static analysis, and theorem proving are playing an increasingly prominent role in the design and analysis of complex hardware and software systems [22,32]. Any given project features workflows that employ multiple tools and techniques. For example, a workflow might use a combination of test case generation and static analysis to identify bugs, synthesis to generate correct code, and verification to establish correctness properties. Since formal techniques are typically used to achieve a high degree of assurance for certifying the validity of a system, it is important to extract evidence for any associated formal and informal claims. This evidence should be explicit so that it can be examined, replayed, and maintained even as the constituent components

* This work was supported by NSF Grant CSR-EHCS(CPS)-0834810, NASA Cooperative Agreement NNX08AY53A, and by DARPA under agreement number FA8750-12-C-0284. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 275–294, 2013.

are modified. The idea of an Evidential Tool Bus (ETB) was proposed and motivated by Rushby [30]. Building on these ideas, we recently designed and implemented an architecture for ETB as a distributed framework for integrating diverse tools into coherent workflows for producing claims supported by explicit evidence. We motivate the need for such an architecture and describe the design, implementation, and application of ETB.

Recent years have witnessed an explosion in the growth of formal verification tools. We now have a range of powerful formal tools for specialized tasks, but a typical application of formal techniques employs multiple tools. Examples of such integrated formal techniques include

1. *Counterexample-guided abstraction refinement (CEGAR)* combining model checking and abstraction, which is itself typically implemented using a satisfiability solver [8].
2. *Concolic execution* combining symbolic evaluation with a constraint solver for generating test cases [31].
3. *Hardware verification toolchains* that exploit different representations including hardware description languages and Boolean representations such as and-inverter graphs, binary decision diagrams, and conjunctive normal form [6].
4. *Deductive program verification* to generate proof obligations that are then discharged using a theorem prover [25].
5. *Invariant generation* combining static analysis, templates, dynamic analysis, k-induction, and property-directed reachability [9,15,5].

These examples employ several tools within a script to produce formal claims that either verify software properties or produce concrete counterexamples. Quite often, these tools have specific platform requirements so that any framework for combining them must support multiple diverse platforms.

Software and system certification is another motivation for ETB as a framework for the construction and maintenance of arguments and evidence. A formally supported software design lifecycle includes several phases from requirements and specification to design and implementation to testing and integration. These phases are connected to each other through formal and semi-formal claims. The resulting system is often accompanied by an assurance case [4], which is *"a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a systems properties are adequately justified for a given application in a given environment."* Such an assurance case is an argument employing claims that are supported by evidence, where some of the evidence is the result of applying individual tools.

The main purpose of ETB is the production of claims supported by arguments based on evidence. Some sub-claims in the argument can be established by external tools. This yields several desiderata that guide the design of the evidential tool bus.

1. ETB should be extensible with new claim forms and rules of argumentation.
2. It should admit new external tools (including human oracles) through an API to produce claims and generate queries.

3. New workflows should be definable as scripts.
4. There should be support for maintaining an argument even as the underlying tools and inputs change.
5. A certifying agent should be able to replay and check any part of a completed development, relative to explicitly stated assumptions.

The most important design principle for ETB is *semantic neutrality*. The framework should not be biased toward any specific tools, languages, models, forms of evidence, or applications. Any semantic interpretation is provided solely by the tools. The ETB merely provides the plumbing and book-keeping so that tools can communicate with each other to exchange services while recording the resulting claims and evidence.

Given these considerations, we have designed ETB as a distributed client-server architecture where the coordination between services is defined using a variant of the Datalog programming language [1,7]. We use Datalog both as a metalanguage for representing claims, inference rules, and arguments, as well as a scripting and coordination language for capturing distributed workflows. Datalog was initially conceived as a database query language, but in recent years, it has been used for other interesting applications such as static analysis, declarative networking, and verification [17,21]. In our variant of Datalog, claims are expressed using Datalog predicates. We incorporate external tool invocations through *interpreted* predicates. These external tools are available as services provided by one or more servers on the ETB network. At these servers, tool *wrappers* are defined to map Datalog queries of these predicates to the corresponding tool invocation and bind the results back to variables in the query. A typical interpreted query would be $minisatCheck(Formula, Result)$, which invokes the MiniSAT [14] SAT solver on the given formula to bind the variable *Result* to either `sat` or `unsat`. Claims can also be expressed using uninterpreted predicates. For example, we might have a more generic satisfiability query given by $satisfiable(Formula, Result)$ which can be defined to invoke one of several SAT solvers such as MiniSAT. Datalog programs are defined as Horn clauses, where the head atom is expressed using an uninterpreted predicate. A subset of these program clauses can be identified as *inference rules* that are admissible as steps within an argument, while the remaining clauses are used as scripts to direct the workflow.

ETB features a distributed architecture for processing Datalog queries. An ETB network is a fully-connected graph of ETB servers operating on a local area network. ETB nodes or networks running outside of the local network can be integrated through proxy servers. Each server advertises the tool services that it offers on the network. The ETB network offers a client API that can be used to define, initiate, and monitor computations on the ETB network as well as to maintain the structure of the network itself. An ETB network can be used simultaneously by several clients. Each server also maintains a claims table and a version-controlled file system. The claims table monitors the status of queries and records claims together with the associated inference steps. The claims are about data, some of which are maintained in files. It is important to note that the claims are actually about (the contents of) file versions, so that a claim about one version may not

be valid about other versions of the same file. ETB can be used to maintain the validity of claims and arguments as file versions are modified.

Failure is a pervasive problem with distributed applications, and ETB is no exception. ETB server nodes can go down, communication links can fail, and tool invocations can trigger errors. Such erroneous computations might corrupt the data in files. In ETB, errors generate *error claims* that can be logged or used to trigger notification and recovery actions.

ETB has applications beyond supporting workflows for formal verification and certification. ETB can be used as a middleware platform for orchestrating distributed computations that involve the production and maintenance of claims. Examples include any scientific workflow where it is useful to maintain information about the provenance of files. Another such application is a distributed `make` (described in Section 4) that keeps system builds synchronized with changes to the source code files. Here, claims about the relationship between source and binary files, as well as configuration details, are used to direct the build process. Unlike the Unix utility `make`, the ETB version is sensitive only to file contents and not their write dates. Another application is in tying together tools that run only on specific platforms, e.g., Windows, Linux, iOS, and Android. ETB can also be used to distribute tasks such as regression testing that can be easily parallelized and distributed across a network of servers.

It is important to identify the problems that are not addressed by ETB. It is meant for coarse-grained integration of tasks over a distributed network, and not for fine-grained integration as in a Nelson–Oppen combination [26] of inference procedures. The integration in ETB is especially suited to workflows where the individual tools are employed on large tasks where the inputs and outputs can be saved to and read from files. In such workflows, much of the data saved in files is part of the evidence that has to be preserved along with any associated claims for certification purposes. While ETB can be used as a general-purpose framework for coarse-grained distributed computing, the main advantage it offers over task distribution frameworks like Condor [33] or Hadoop is its ability to retain evidence together with reproducible claims. Unlike Programatica [19], which embeds certificates into Haskell programs, ETB is a general-purpose distributed framework for integrating tools as well as evidence. ETB, by itself, is not a semantic interchange framework like PROSPER [12], ToolBus [11], Veritech [18], Boogie [3], Why [16], or Frama-C [10] where different tools interact through a common interchange language. However, ETB could be used to implement such frameworks or to work in conjunction with them. ETB has some similarity to service-oriented architectures such as ETI/JETI [24] or Orc [23] which can also be used to define workflows, but unlike ETB, these do not support the construction of version-controlled, evidence-based arguments. Dedalus [2] is a time and space-aware declarative language that extends Datalog for declarative networking and computation. It explicitly embeds a discrete notion of time to perform atomic updates on the set of extensive set of atoms considered true at any moment. Unlike ETB, which supports the exchange of files and tool services, the operational semantics of Dedalus only supports the exchange of Datalog atoms between nodes ("locations"). ETB is also different from metalogical frameworks

like LF [20], Isabelle [27], and Twelf [28]. These frameworks are used to represent formal object logics and support proof construction. In contrast, ETB is neutral about syntactic representations and semantic interpretations and focuses purely on the interaction between tools through scripts, queries, and claims. The design of ETB has been influenced by ideas from these projects, but the focus has been on semantically neutral, evidence-based tool integration.

We outline the use of Datalog as the metalanguage of ETB in Section 2, and describe the ETB architecture in Section 3. In Section 4, we illustrate the workings of ETB with some examples, and conclude with a discussion of future work in Section 5.

2 Datalog as a Metalanguage

Datalog was initially introduced as a deductive database language, but in recent years, it has been employed as a declarative framework for other applications such as program analysis and networking. We describe how ETB uses a variant of Datalog to represent data, claims, queries, inference rules, workflows, and arguments. We illustrate our variant of Datalog with an example. The key ideas in the representation of ETB artifacts are

1. Data in the ETB consists of file handles, tool session handles, as well as JSON data objects (Booleans, numbers, strings, structures, or arrays).
2. An ETB atom is an n-ary predicate applied to n arguments that are either data objects or variables.
3. A claim is a ground (i.e., fully instantiated) atom or its negation.
4. A query atom is a partially instantiated atom where some of the arguments can be variables. A query is a sequence of query atoms.
5. Information in the ETB is exchanged in the form of *queries* and *claims* about these data objects. Files related to the queries and claims are also exchanged across the ETB network.
6. External tools are integrated into ETB through *interpreted* predicates.
7. Scripts are ETB programs built from *derivation rules*, i.e., Horn clauses where the head atom contains an *uninterpreted* predicate.
8. *Inference rules* are a subset of the derivation rules that are used to construct *exportable* proofs of claims from those of sub-claims. Such proofs may need to satisfy constraints on the subset of external tools and rules that can be used.
9. An ETB proof is a tree of claims: each claim follows from the subclaims by an inference rule.
10. A query triggers backward chaining on derivation rules to instantiate a workflow. A successful computation yields a set of claims instantiating the query, and each such a claim is supported by one or more derivations.
11. Forward chaining is employed on inference rules to derive claims and construct exportable proofs.
12. Multiple derivations (including exportable proofs) can be saved. These derivations are used to recreate claims when the input data has changed. Multiple exportable proofs are also useful in strengthening the validity of a claim.

We describe each of these aspects of ETB's use of Datalog as a metalanguage.

Data. The data processed by ETB can include programs, transition systems, formulas, files, contexts, models, test cases, analysis results, and tool sessions. ETB provides no built-in interpretation of the data beyond what is provided by these tools. Among the tools in the ETB are translators from one data representation to another. For example, an ETB tool might translate formulas generated by a hardware description language to the input language of a model checker. ETB data is represented as a JSON object. File and tool handles are represented as JSON structures. File handles can contain metadata, but are uniquely identified by the hash of the file contents. This ensures that any claims about files are about their contents rather than their file IDs or other metadata. Tool sessions are also similarly identified along with any metadata, but are uniquely identified by a session ID and a hash of the session state that is maintained by the tool wrapper. ETB can be extended with new forms of data, but the above forms are sufficient for most applications.

Claims. The notion of a claim is central to ETB. The whole point of ETB is to produce a replayable argument for a claim that is supported by evidence. Such an argument can include sub-claims generated by external tools. Claims are given in the form of judgements that are applied to data. A claim is a ground ETB atom of the form $p(d_1, \ldots, d_n)$, where p is an n-ary predicate and $d_1, \ldots, d_n$ is a sequence of n data elements. A predicate p can either be an *uninterpreted* predicate or an *interpreted* predicate. The latter class of predicates is mapped to tool invocations, whereas the former corresponds to conventional Datalog predicates. In describing these claims, we make reference to external tools such as the PVS theorem prover, the Yices SMT solver, and the SAL model checker. Simple examples of judgements used to make claims include:

1. $pvsFormula(P)$: P is a PVS formula
2. $pvs2Yices(P, Y)$: Y is the Yices translation of PVS formula P
3. $yicesModel(Y, C)$: C is a model for Yices formula Y
4. $yicesUnsat(Q)$: Q is an unsatisfiable Yices context
5. $salModule(M)$: M is a SAL transition system module
6. $invariant(M, P)$: P is an invariant property of module M

The actual claim predicates might take additional arguments such as tool version or auxiliary arguments.

Most of the predicates in the above list of claims are interpreted, i.e., they are directly implemented by invoking external tools. A predicate like *invariant* is uninterpreted in that it is defined by means of ETB rules. In Datalog, the predicates are partitioned as (extensional) database predicates and (intentional) defined ones. In ETB, the extensional predicates are treated as interpreted since the database can be regarded as an external tool. Another side-effect of using interpreted predicates is that we can recover the power of Prolog-style unification through external tool invocations. Datalog treats the data elements as unstructured, whereas Prolog can use unification to decompose a term of the form $f(X, Y)$. With interpreted predicates, we can replace a goal formula of the form $p(f(X, Y))$ by $p(Z), decompose_f(Z, X, Y)$, where the interpreted predicate

decompose$_f$ holds exactly when $Z = f(X, Y)$. A program clause of the form $p(f(X, Y))$:- $q(X, Y)$ can be replaced by $p(Z)$:- *decompose*$_f(Z, X, Y), q(X, Y)$.

Claims in ETB are not restricted to logical assertions. For example, a claim can provide statistical information such as the percentage of coverage provided by a test suite. Claims can even be speculative, as in the assertion that a formula is a potential invariant as generated by a tool like Daikon [15].

Queries. A query is a non-empty sequence of partially instantiated ETB atoms of the form $p(a_1, \ldots, a_n)$ where some arguments a_i can be variables. If $\{X_1, \ldots, X_n\}$ is the set of free variables in the query sequence $A_1, \ldots, A_k$, and K is the Datalog program, then a Datalog computation searches for a set of instantiations of the form $[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n]$, where each b_i is a data object, such that $K \implies A_1[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n] \wedge \cdots \wedge A_k[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n]$. When this is the case, we can assert $A_i[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n]$ as a claim, for i in $\{1 \ldots k\}$.

Queries in ETB are used to trigger computations to establish the corresponding claims. Typical queries might include

1. Translation from one language, e.g., PVS to Yices: *pvs2Yices*$(`x+y < 3', Y)$
2. Type Checking: *pvsTypecheck*$('x : real, y : int', '(x + y)/(x - y)', T, Q)$, where the first argument is the context, and the output T is the type of the expression, and output Q is the set of proof obligations, e.g., $x - y \neq 0$.
3. Satisfiability checking: *yicesUnsat*(*filehandle*), where the file handle is provided by some other tool, e.g., the translator from PVS.

Scripts. An ETB program is collection of Horn clauses, where each clause is either a derivation rule of the form H :- C or an inference rule of the form $H \Leftarrow C$, where H is an atom, and C is a list of atoms such that every free variable in H also occurs free in some atom of C. In ETB, the predicate symbol for the head atom, the head predicate, must be uninterpreted, and the set of clauses in which p occurs as the head predicate constitutes the definition of p. As in Prolog and Datalog, variables are represented by identifiers where the leading character is in upper case, whereas the leading character for identifiers for constants is in lower case. For example, one could write the following reachability script for a transition system using a BDD package that checks if some bad state satisfying P is reachable by a transition system M. In this script, we first extract the initialization predicate I and the transition relation N from module M, and then check if P is reachable with I and N. The predicate *reachable* is defined to construct the BDD versions of I, N, and P as L, W, and Q, and to invoke the predicate *bddReachable* on these. The latter predicate is defined to iteratively compute the image of W with respect to L as J, and then check the reachability of Q from J.

$$\begin{aligned}
\mathit{reach}(M, P) &\text{ :- } \mathit{init}(M, I), \mathit{transition}(M, N), \mathit{reachable}(I, N, P).\\
\mathit{reachable}(I, N, P) &\text{ :- } \mathit{bdd}(I, L), \mathit{bdd}(N, W), \mathit{bdd}(P, Q), \mathit{bddReachable}(L, W, Q).\\
\mathit{bddReachable}(L, W, Q) &\text{ :- } \mathit{bddAnd}(L, Q, R), \mathit{bddNonempty}(R).\\
\mathit{bddReachable}(L, W, Q) &\text{ :- } \mathit{bddImage}(W, L, J), \mathit{bddReachable}(J, W, Q).
\end{aligned}$$

Semantics. We describe the distributed evaluation of ETB queries in Section 3.2. We focus here on the semantics of the Datalog variant used by ETB. With external tool invocations, the data domain is no longer finite. For example, a simple successor relation $succ(X, Y)$ which holds exactly when $Y = X + 1$ would generate an infinite set. We assume that the interpreted queries are *bounded*, so that $p(a, X)$ binds X to a list of instantiations for X. The processing of interpreted queries through a tool wrapper can trigger back-chaining, so that the query $p(a, X)$ can introduce the query $q(b)$. Our semantics is proof-theoretic. Given a program Π and a query atom Q, the meaning $\Pi[\![Q]\!]$ consists of the set of ground instances $\underline{Q}$ of Q such that there is an SLD-resolution proof of $\underline{Q}$ from Π. There can be unboundedly many such instances, as for example for the query $lt(0, Y)$ in the program below.

$$\begin{aligned} lt(X, Y) &\text{ :- } succ(X, Y). \\ lt(X, Y) &\text{ :- } lt(X, Z), lt(Z, Y). \end{aligned}$$

In such cases, the computation diverges.

Inference Rules. Some of the clauses in the ETB programs can be designated as inference rules that are sanctioned for use in ETB proofs. For example, in the inference rules below, P is asserted to be an invariant for a transition module with initialization I and transition relation N, if it is implied by another invariant, or it is an inductive invariant. The premises of the rule involving the predicates *implies* and *inductiveInvariant* can be established by other rules or by means of tool invocation.

$$\begin{aligned} invariant(I, N, P) &\Leftarrow invariant(I, N, Q), implies(Q, P). \\ invariant(I, N, P) &\Leftarrow inductiveInvariant(I, N, P). \end{aligned}$$

Inference rules are ETB program clauses and can also be used as scripts. Other examples of inference rules include

1. Counterexample traces: A valid trace is one that leads from an initial state to an error state on zero or more computation steps.
2. Abstraction: If one program is an abstraction of a (concrete) program, then the concrete counterpart of a property of the abstract program holds of the concrete one.
3. Refinement: A concrete program is a refinement of an abstract one if any concrete computation step can be simulated by the abstract program through the simulation relation. Separate inference rules can be specified for the different forms of refinement.
4. Termination: A `while`-loop terminates if there is a ranking function on the variables that decreases with each execution of the loop. Other termination techniques, e.g., disjunctive well-foundedness of the transitive closure of the transition relation, can also be specified [17].

Proofs. A proof is a derivation, a tree whose root node is a claim that is an instance of the head atom of an inference rule, where the corresponding instances

of the body atoms are proved by the subtrees. Each ETB claim that has been established

Typically, an interpreted claim is the leaf node of a proof, but it is possible for the proof of an interpreted claim to also have sub-proofs corresponding to queries dynamically generated during the tool invocation. In this case, the validity of the inference step relative to the sub-proofs is checked by the tool itself.

In some cases, the tool might be somewhat nondeterministic so that the proof cannot be exactly replicated. In such cases, the proof is merely used as a template for reconstructing the steps in the justification. ETB proofs are thus a subset of ETB derivations where the proof steps only employ program clauses that are designated as inference rules. The proof construction feature has not yet been implemented in ETB.

2.1 An Example: Iterated *k*-Induction

Given a transition system $M = \langle I, N \rangle$ with initialization I and transition N, and a predicate P, we want to prove that P is an invariant by finding the smallest k such that P is k-inductive. The inference rule for demonstrating that P is an invariant of M is given below. It asserts that one possible way of demonstrating the P is an invariant of M is by establishing that P is a k-inductive. (There are other possible ways, such as by showing that P is entailed by some other invariant Q.)

$$\begin{aligned} \mathit{invariant}(M, P) \Leftarrow\ & \mathit{transitionSystem}(M), \\ & \mathit{initialization}(M, I), \\ & \mathit{transition}(M, N), \\ & \mathit{kInductive}(I, N, P, K). \end{aligned}$$

The inference rule for k-inductivity decomposes the claim into the subclaims that P holds for the first K steps (i.e., the bounded model checking claim *bmc* holds of I, N, P, K), and that if P holds for any K states in a computation, then it holds for the $K + 1$st state of the computation.

$$\begin{aligned} \mathit{kInductive}(I, N, P, K) \Leftarrow\ & \mathit{bmc}(I, N, P, K), \\ & \mathit{inductionStep}(N, P, K). \end{aligned}$$

The above inference rules are used to build ETB proof, but the actual value of K is found by a script defining the predicate *iterInductive* that tries each value K starting from 1 up to some upper bound U using the tool invocation $\mathit{in\_range}(K, 1, U)$. We can also define a script for proving $\mathit{invariant}(M, P)$ by invoking the script for *iterInductive* to find a suitable K for some fixed upper bound, e.g., 5.

$$\begin{aligned} \mathit{iterInductive}(I, N, P, K, U) \text{ :- } & \mathit{in\_range}(K, 1, U), \mathit{kInductive}(I, N, P, K). \\ \mathit{invariant}(M, P) \text{ :- } & \mathit{transitionSystem}(M), \\ & \mathit{initialization}(M, I), \\ & \mathit{transition}(M, N), \\ & \mathit{iterInductive}(I, N, P, K, 5). \end{aligned}$$

The predicates *bmc* and *inductionStep* invoke Yices, by first invoking an interpreted predicate *yicesBmcFormula* to construct the bounded model checking query formula F, and then invoking the interpreted predicate *yicesUnsat* to determine if F is unsatisfiable using Yices.

$$\begin{aligned} bmc(I, N, P, K) \Leftarrow\ & yicesBmcFormula(I, N, P, K, F), \\ & yicesUnsat(F). \end{aligned}$$

The induction step generates the k-induction formula F using the interpreted predicate *yicesInductionFormula*, and checks for unsatisfiability invoking the interpreted predicate *yicesUnsat* as above.

$$\begin{aligned} inductionStep(N, P, K) \Leftarrow\ & yicesInductionFormula(N, P, K, F), \\ & yicesUnsat(F). \end{aligned}$$

We omit the details of the construction of the Yices formulas for BMC and k-induction. If $kInductive(I, N, P, k)$ succeeds for some k, then forward chaining on the rule of inference for *invariant* yields the claim $invariant(M, P)$ with a proof using the inference rules.

3 The Architecture of ETB

ETB is based on a distributed client-server architecture. An ETB cluster consists of a fully connected network of ETB servers as shown in Figure 1. Each server maintains a version controlled file system, specifically Git, and tables of claims and queries. It also advertises a set of services and subscribes to a list of claim forms. A server can itself be a proxy for other ETB clusters, thus serving as a bridge between two or more ETB clusters. An ETB cluster is operated through a client which offers an application programming interface (API). Through a client, a user can configure and program the server, connect other servers to the cluster, add new content, initiate the processing of queries, and monitor the progress of a computation.

We describe some of the key features of the ETB architecture: the software stack, the server, the client API, and the mechanics of distributed query processing in ETB. The current ETB prototype is implemented in Python.

3.1 The ETB Software Stack

The ETB stack consists of the following layers:

1. **Network:** Maintains the connectivity status of the ETB cluster and shares information about tool wrappers and cluster-wide ETB programs.
2. **File:** Each server operates a Git repository from which files/directories are synchronized with servers as needed.
3. **Session:** Servers exchange claims and queries, as needed to invoke external tools.

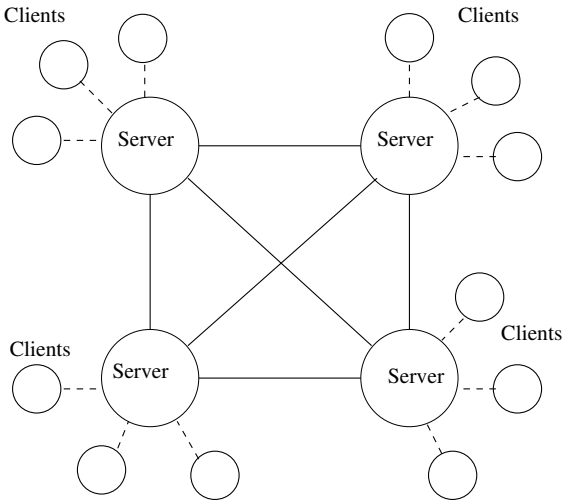


Fig. 1. The ETB Client-Server Architecture

The Network Layer. Basic to the network layer is a heartbeat. Each server periodically broadcasts a heartbeat message to the other servers in the network. This heartbeat is used to maintain the membership of the network; in case of a server failing, other servers will delete it from the cluster. Servers also share information about the predicate symbols they can interpret, e.g., a server on a Linux machine on which Yices [13] is installed may declare that it can interpret *yicesUnsat*.

When two servers are in the same cluster, they can access one another through the network. XML-RPC calls are used for all communication between servers. For instance, the heartbeat messages are remote invocations of a `ping()` method.

The File Layer. Each server has a local, private Git repository that it uses to store files involved in claims. If several versions of a file are involved in claims, each such version is stored, and referred to by its unique SHA1 hash. The purpose of this storage is two-fold:

1. To replay derivations and proofs of claims, which may require restoring old versions of files
2. Sharing files with other servers by extracting the content of the file (indexed by its SHA1 hash) and sending it through the network layer.

Using Git in this way ensures the *immutability* of claims involving the contents of files. If we state that `yicesUnsat`(*filehandle*), and want to replay it later, the file may have been deleted or modified. However, since the contents of the file is saved in the Git repository, and the claim actually contains the hash of the file, we can create the original file again to replay Yices on it. File contents can therefore be seen as constant values indexed by their SHA1 hash even though the file itself might have been modified in the repository.

The Session Layer. The session layer supports query processing in order to produce claims and evidence. Each server executes scripts locally and only invokes the other servers for external tools. The queries corresponding to the tools are sent to a suitable server using XML-RPC. The target server uses the File layer to obtain the relevant files. The processing of the query at the target server can recursively invoke other queries on the ETB network. Any resulting claims are returned to the source server along with the derivation. The claims are also broadcast to servers that have registered subscriptions associated with this claim. The source server then obtains the files associated with the resulting claims. The source server can choose a target server based on load parameters or other factors. If the target server fails during the processing of a query, the query can be retargeted to a different server. Each server has a consistent copy of the ETB scripts, but a server can also have local scripts where only the service provided by these scripts is exported.

3.2 Query Processing in ETB

We present the details on the operational semantics of query processing through the evaluation of ETB rules. A *goal* is an ETB atom, possibly with variables. *Claims* are ground atoms with an associated *derivation*. The *query processing rules* have the form: $\mathcal{R}, T, \mathcal{S} \Downarrow T', \mathcal{S}'$, meaning that a set of ETB rules $\mathcal{R}$, a set of established claims T, and a set of processing clauses $\mathcal{S}$, produce in one step a new set of established claims T' and a new set of processing clauses $\mathcal{S}'$. Established claims in T are pairs of a ground claim h and a justifying clause h :- $g_1, ..., g_n$. Processing clauses in $\mathcal{S}$ are pairs of a justifying clause and a clause (the justifying clause will be used to turn derived unit clauses into claims). Claims that are already in the claims table $\mathcal{T}$ are reused, as in tabled logic programming [29], and are not recomputed.

Figure 2 contains a set of *inference rules* that are applied to the query set $\mathcal{S}$ and the claims table T in a nondeterministic order until no rule affecting $\mathcal{S}$ and $\mathcal{T}$ is applicable; the set $\mathcal{S}$ is then *saturated* and contains clauses and justifications that can be used to construct derivations from the rules supporting the claims in $\mathcal{T}$.

- The rule Claim transfers unit clauses to the claims table.
- The rule Application instantiates a rule from $\mathcal{R}$ to the set $\mathcal{S}$ if the rule may help solving the first goal (g_1) of some clause in S.
- The rule Resolution removes the first goal g_1 if $g_1\sigma$ is a claim. It is really a unit resolution inference rule. Note that the justification $c\sigma[c']$ captures the resolution step.
- The rule Interpretation schedules the interpretation of an interpreted atom g_1 if it occurs as the first premise of a clause in $\mathcal{S}$. Intuitively, a tool invocation solves this goal in the hope that it will add claims matching g_1, that will in turn remove g_1 by the Resolution rule. All claims matching g_1 generated by the interpretation I are added to T when all the claims in Q corresponding to recursive queries have been established in T. A tool can also introduce additional claims H to T, and each additional claim f is

$$\textsc{Claim}\quad \frac{(c : h \text{ :- }) \in \mathcal{S}}{\mathcal{R}, T, \mathcal{S} \Downarrow T + (c : h), \mathcal{S}}$$

$$\textsc{Application}\quad \frac{(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \qquad r \equiv a \text{ :- } b_1, ..., b_k \in \mathcal{R} \qquad \exists\sigma. a\sigma = g_1\sigma}{\mathcal{R}, T, \mathcal{S} \Downarrow T, \mathcal{S} + (r\sigma : a\sigma \text{ :- } b_1\sigma, ..., b_k\sigma)}$$

$$\textsc{Resolution}\quad \frac{(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \qquad \exists\sigma.(c' : g_1\sigma) \in T}{\mathcal{R}, T, \mathcal{S} \Downarrow T, \mathcal{S} + (c\sigma[c']) : h\sigma \text{ :- } g_2\sigma, ..., g_n\sigma)}$$

$$\textsc{Interpretation}\quad \frac{(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \qquad g_1 \text{ interpreted} \qquad I(g_1) = \Sigma, H, Q \qquad Q \subseteq T}{\mathcal{R}, T, \mathcal{S} \Downarrow T + \{(f \text{ :- }^{g_1} Q) : f | f \in H\} + \{(g_1\sigma \text{ :- } Q) : g_1\sigma | \sigma \in \Sigma\}, \mathcal{S}}$$

$$\textsc{Query}\quad \frac{\begin{array}{c}(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \\ g_1 \text{ interpreted} \qquad q \text{ query generated during interpretation of } g_1\end{array}}{\mathcal{R}, T, \mathcal{S} \Downarrow T, \mathcal{S} + (g_1 : q)}$$

Fig. 2. Rules for query processing

tagged with the derivation $(f \text{ :- }^{g_1} Q)$ to indicate that it was derived during the invocation g_1 and is conditional on Q.
- The rule Query allows new queries to be added to $\mathcal{S}$ during the evaluation of an interpreted atom g_1 by the corresponding external tool.

This calculus is trivially sound (the rule Resolution is an instance of unit resolution). It is also complete for well-formed first-order clauses (such that all variables of the head of a clause also occur in the body). Any SLD-resolution refutation proof can be transformed to a derivation in the ETB calculus.

3.3 Tool Wrappers

Tools are connected to the ETB by means of thin wrappers. Wrappers are written in Python and are short, simple programs that map queries to tool invocations. A wrapper for a tool is a Python class that defines one or more predicates to be interpreted by the ETB. Predicates are given by a triplet: a name, a predicate signature, and a Python function. The name and signature of the predicate are shared among all ETB nodes, informing other nodes that the ETB can now interpret this predicate. The function is local, and is executed when the predicate needs to be interpreted.

Predicate Signatures. A predicate signature is defined by the following grammar:

predicate signature	$sig ::= arg\_spec \mid sig, arg\_spec$
argument specification	$arg\_spec ::= sign\ arg : type$
argument type	$type ::= \texttt{value} \mid \texttt{file} \mid \texttt{files}$
argument sign	$sign ::= + \mid -$

The signature associate modes and types to each argument of the predicate. Types are either `value`, which indicates a JSON value, or `file`, which indicates an ETB file references, or `files`, in which case the argument is a list of files. The mode associated to each argument in the signature represents inputs and outputs: a + argument is an input to the predicate and is required to be ground when the predicate is invoked. A - argument indicates an output and can be either a constant or a variable at invocation: if it is a constant, the predicate is expected to check that the predicate holds for this particular value, if it is a variable, the predicate is expected to return one or more substitutions for this variable.

For example, the signature for a predicate invoking the `yices` SMT solver on a file containing a formula and returning either of the string "sat" or "unsat" could be:

```
yices :: +input: file, -result: value
```

The predicate *yicesUnsat* used earlier (page 284) can be defined as a separate wrapper or as an uninterpreted predicate using the `yices` predicate.

Implementation of a Wrapper An ETB tool wrapper is a Python class that inherits from the class `etb.wrapper.Tool`. This class provides a decorator `predicate` which is used to declare particular methods of the class as predicates as well as give the predicate signature. A template wrapper for the `yices` SMT solver is the following:

```
class Yices(Tool):
    @predicate('+input: file, -result: value')
    def yices(self, formula, result):
      ...
```

It declares the predicate `yices` with the signature we saw previously.

The code of the wrapper then typically calls the tool on the input arguments, gets the results from the tool and translates them for the ETB. The results returned to the ETB when the query succeeds, include a (possibly empty) list of substitutions, additional ground claims, and claims associated with ETB queries dynamically generated during the tool invocation.

The `etb.wrapper.Tool` class provides utilities that help in interpreting and producing ETB values. The `etb.wrapper.BatchTool` class extends it with utilities to make it easy to call external batch tools. Using these utilities, we can write the code of the `yices` wrapper as follows:

```
@Tool.predicate("+input: file, -result: value")
def yices(self, input, result):
    return self.run(result, 'yices', input['file'])
```

The wrapper takes as output argument a value `result`, which can be either a constant or a variable. If it is a constant, the wrapper checks that `yices` produces the same value, and returns an empty substitution representing success. If it is a variable, it returns a substitution binding the variable to `yices` output. If the `yices` command fails for any reason (e.g. `yices` is not found, or the file is not a valid `yices` file, etc.), it returns failure (no substitution), as well as an additional error claim.

3.4 The Client API

Interaction with an ETB network is done through a client. The client can exchange files with the ETB repository. It can also initiate queries, monitor the progress of a query, and access the substitutions and claims established by query evaluation.

The basic client included with ETB is a read-eval-print interaction loop which can interpret scripts interacting with the ETB. Specialized clients can be built for specific applications — a GUI client has been implemented for a hardware analysis project. We have also defined a shell client for accessing ETB.

A client connects to one server of an ETB network, and communicates with that node using XML-RPC calls. Clients can therefore be written in any language that supports XML-RPC. The main functions of the client API to the ETB are presented in Figure 3 – other functions not listed here enable the client to manage the state of the ETB and observe a visual representation of running queries.

Method	**Description**
ref = `put_file`(*src*, *dst*)	Put the file *src* as *dst* on the ETB, and return a reference to the destination.
`get_file`(*srcref*, *dst*)	Get the file corresponding to the reference *srcref* from the ETB and copy it locally under *dst*.
id = `eval`(*claim*)	Create a new query looking for solution for *claim*, and return a query ID.
answers = `wait_query`(*id*)	Wait for the query *id* to complete and returns its set of answers.
answers = `query_answers`(*id*)	Return the current set of answers for the query *id*.
claims = `query_claims`(*id*)	Return the current set of claims generated as part of the query *id*.
ret = `connect`(*host*, *port*)	Connect the remote ETB node running at *host:port* with the local ETB.

Fig. 3. The ETB Client API

4 ETB Examples

In this section, we present two simple application of the ETB. The first is a distributed implementation of `make` that can also handle multiple platforms,

and the second is a naïve AllSAT script. ETB is currently being used in a couple of projects: one, to integrate various hardware analysis tools, and another, to combine model-based design/analysis capabilities.

4.1 A Simple Distributed Make

In our first example, we use the ETB to build a binary executable from source C files as would typically be done using `make` or similar tools.

C Compiler Wrappers We first define a wrapper for `gcc`. This wrapper declares two predicates, one to compile a C source file into an object file, the other one to link several object files together into a binary.

```
@Tool.predicate("+src: file, +dependencies: files, -obj: file")
def gcc_compile(self, src, deps, obj):
   dst = os.path.splitext(src['file'])[0] + '.o'
   self.callTool('gcc', '-c', '-o', dst, src['file'])
   objref = self.fs.put_file(dst)
   return self.bindResult(obj, objref)
```

The first wrapper takes a source file, a list of files that the compilation depends on, typically header files, and returns an object file. We first create the output file name, then run `gcc`. Finally, we put the generated object file on the ETB and bind the output variable to this file. Note that the dependencies do not appear in the command that is run, but appear in the claim, and as such the files are copied to the server that handles the compilation together with the source file.

```
@Tool.predicate("+ofiles: files, +exename: value, -exe: file")
def gcc_link(self, ofiles, exename, exe):
   filenames = [ r['file'] for r in ofiles ]
   args = [ 'gcc', '-o', str(exename) ] + filenames
   self.callTool(*args, env=env)
   exeref = self.fs.put_file(str(exename))
   return self.bindResult(exe, exeref)
```

The second predicate is similar to the first one. It takes a list of objects files and produces a binary executable.

A rule to build a program. We can now use these wrappers to build a program. Our example is built from three C source files: two independent components with associated header files, and a main program using these two components.

The following rule compiles each source file in turn, then links them:

```
main(Src1, H1, Src2, H2, Main, Name, Exe) :-
  gcc_compile(Src1, [H1], Obj1),
  gcc_compile(Src2, [H2], Obj2),
  gcc_compile(Main, [H1, H2], ObjMain),
  gcc_link([Obj1, Obj2, ObjMain], Name, Exe).
```

The rule defines a predicate `main` which takes as argument the three source files and two header files. It also takes the name of the binary and the binary itself.

To build an actual binary, we just need to make an ETB query applying actual files to the rule.

Our example does not take into account the architecture on which the ETB node is running, or the version of the compiler used. This kind of information, as well as additional compilation flags, or other metadata can be tracked using the ETB.

4.2 AllSAT on Top of Yices

In our second example, we build a tool that can generate all satisfying truth assignments for a formula using a SAT solver. The idea is straightforward: ask the solver if the formula is satisfiable, if it is, ask the solver to generate a solution, add a clause blocking this solution to the formula, and iterate. Each iteration finds a new solution until they have all been found.

Assuming that we have two predicates `sat` and `unsat` that can establish whether a formula is satisfiable, and return a solution if it is, we can write derivation rules for a predicate `allsat` as follows:

```
allsat(F, Answers) :- sat(F, M),
                      negateModel(F, M, NewF),
                      allsat(NewF, T),
                      cons(M, T, Answers).
allsat(F, Answers) :- unsat(F), Answers = nil.
```

We use the additional predicate `negateModel` that extends an existing formula with a blocking clause built from a solution, as well as utility predicates `cons` and `nil` to build the list of answers and `equal` to test for equality. `cons` and `nil` are interpreted predicates that bind their last argument to new identifiers (similar to dynamic allocation that "creates" new memory addresses) to identify list nodes.

Given a solver, and its chosen representation of a formula, we can instantiate the predicates `sat`, `unsat`, and `negateModel` to get a working implementation of `allsat`. In this example, we use the `yices` SMT solver, and its native input language. We write a single interpreted predicate that calls the solver on a file containing a formula, and returns either "sat" or "unsat" as well as a model if available:

```
@Tool.predicate("+formula: file, -result: value, -model: value")
def yices(self, formula, result, model):
    ...
```

From this, we can derive the two predicates `sat` and `unsat`:

```
sat(F, M) :- yices(F, S, M), equal(S, 'sat').
unsat(F) :- yices(F, S, M), equal(S, 'unsat').
```

5 Conclusions and Future Work

We have outlined our vision for the Evidential Tool Bus, and described the design, implementation, and application of the framework. We have implemented

the basic components of our proposed design, including the server engine, the client and wrapper APIs, wrappers for Yices and SAL, and several small scripts. We have also built capabilities for single-stepping the execution and observing execution traces. A lot of work remains to be done in extending the implementation, developing applications, and optimizing the implementation based on these applications. These include

1. Wrappers for a range of verification tools and static and dynamic analysis tools, through the use of standardized languages front-ends. Many of these wrappers will, we hope, be contributed by third-party developers.
2. Scripts for abstraction, invariant generation, termination, test generation, code generation, synthesis, and result certification.
3. Support for ETB proofs using forward-chaining on inference rules.

We also plan to develop a network of ETB servers operating in the cloud, and to aggressively explore novel applications of tool integration and coordination that require the construction and custody of evidence.

In summary, ETB is a semantically simple framework for formal tool integration that yields replayable evidence for use in assurance cases. It uses a variant of Datalog as the metalanguage for representing claims, queries, workflows, and arguments. Tools are integrated as uninterpreted predicates through wrappers, and made available through a client-server network. Claims are established through distributed query processing over this network, and proofs are constructed from these claims. ETB will be made available in open source form for community-wide experimentation and enhancement.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Alvaro, P., Marczak, W., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.C.: Dedalus: Datalog in Time and Space. Technical report, EECS Department, University of California, Berkeley (December 2009)
3. 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)
4. Bloomfield, R.E., Bishop, P.G., Jones, C.C.M., Froome, P.K.D.: Adelard Safety Case Development Manual. Adelard (1998)
5. Bradley, A.R.: Understanding IC3. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 1–14. Springer, Heidelberg (2012)
6. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
7. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1), 146–166 (1989)
8. 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)

9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
10. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A Software Analysis Perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
11. de Jong, H., Klint, P.: ToolBus: The Next Generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 220–241. Springer, Heidelberg (2003)
12. Dennis, L.A., Collins, G., Norrish, M., Boulton, R., Slind, K., Robinson, G., Gordon, M., Melham, T.: The PROSPER Toolkit. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 78–92. Springer, Heidelberg (2000)
13. 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)
14. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming (2006)
16. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
17. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A Software Verifier Based on Horn Clauses- (Competition Contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
18. Grumberg, O., Katz, S.: Veritech: a framework for translating among model description notations. STTT 9(2), 119–132 (2007)
19. Hallgren, T., Hook, J., Jones, M.P., Kieburtz, R.B.: An overview of the Programatica toolset. In: High Confidence Software and Systems Conference, HCSS 2004 (2004)
20. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. In: IEEE Symposium on Logic in Computer Science, Ithaca, NY (1987)
21. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: Sellis, T.K., Miller, R.J., Kementsietsidis, A., Velegrakis, Y. (eds.) SIGMOD Conference, pp. 1213–1216. ACM (2011)
22. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys 41(4), 21:1–21:54 (2009)
23. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc Programming Language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
24. Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in JETI. In: ECBS, pp. 431–436. IEEE Computer Society (2005)
25. Nelson, G.: Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca. (1981)
26. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
27. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002), Isabelle home page: http://isabelle.in.tum.de/

28. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE-16. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
29. Rao, P., Sagonas, K.F., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Effciently Computing WFS. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 430–440. Springer, Heidelberg (1997)
30. Rushby, J.M.: An Evidential Tool Bus. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 36–36. Springer, Heidelberg (2005)
31. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H. (eds.) Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2005, Lisbon, Portugal, September 5-9, pp. 263–272. ACM (2005)
32. Shankar, N.: Automated deduction for verification. ACM Computing Surveys 41(4), 20:1–20:56 (2009)
33. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. Concurrency - Practice and Experience 17(2-4), 323–356 (2005)

Compositional and Lightweight Dependent Type Inference for ML

He Zhu and Suresh Jagannathan

Dept. of Computer Science
Purdue University

Abstract. We consider the problem of inferring expressive safety properties of higher-order functional programs using first-order decision procedures. Our approach encodes higher-order features into first-order logic formula whose solution can be derived using a lightweight counterexample guided refinement loop. To do so, we extract initial verification conditions from dependent typing rules derived by a syntactic scan of the program. Subsequent type-checking and type-refinement phases infer and propagate specifications of higher order functions, which are treated as *uninterpreted* first-order constructs, via subtyping chains. Our technique provides several benefits not found in existing systems: (1) it enables *compositional* verification and inference of useful safety properties for functional programs; (2) additionally provides counterexamples that serve as witnesses of unsound assertions: (3) does not entail a complex translation or encoding of the original source program into a first-order representation; and, (4) most importantly, profitably employs the large body of existing work on verification of first-order imperative programs to enable efficient analysis of higher-order ones. We have implemented the technique as part of the MLton SML compiler toolchain, where it has shown to be effective in discovering useful invariants with low annotation burden.

1 Introduction

Dependent or refinement types [20,10,28] offer a promising way to express rich invariants in functional programs that can go beyond the capabilities of traditional type systems [8] or control-flow analyses [25], albeit at the price of automatic inference. Recently, there has been substantial progress in reducing this annotation burden [15,23,14,13,16,17,19,18,22] using techniques adopted from model-checking and verification of first-order imperative programs [11,7]. These solutions, however, either (a) involve a complex reformulation of the intuitions underlying invariant detection and verification from a first-order context to a higher-order one [17,18], making it difficult to directly reuse existing tools and methodologies, (b) infer dependent types by solving a set of constraints collected by a whole-program analysis [15,23], additionally seeded with programmer-specified qualifiers, that can impact compositionality and usability, or (c) entail a non-trivial translation to a first-order setting [13], making it

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 295–314, 2013.

more complicated to relate the inferences deduced in the translated first-order representation back to the original higher-order source when there is a failure.

In this paper, we present POPEYE, a compositional verification system that integrates a first-order verification engine, unaware of higher-order control- and dataflow, into a path- and context-sensitive dependent type inference framework for Standard ML. Notably, our solution treats uses of higher-order functions as *uninterpreted* terms. In this way, we are able to directly exploit the scalability and efficiency characteristics of first-order verification tools *without* having to either consider a sophisticated translation or encoding of our functional source program into a first-order one [13], or to re-engineer these tools for a higher-order setting [17]. Our verification strategy is based on a counterexample-guided refinement loop that systematically strengthens a function's inferred dependent type based on new predicates discovered during examination of a derived counterexample path. Moreover, our strategy allows us not to only verify the validity of complex assertions, but can also be used to directly provide counterexample witnesses that disprove the validity of presumed invariants that are incorrect.

Our technique is *compositional* because it lazily propagates refinements computed at call-sites to procedures and *vise versa*, allowing procedure specifications to be strengthened incrementally. It is *lightweight* because it directly operates on source programs without the need to generate arbitrary program slices [26], translate the source to a first-order program [13], or abstract the source to a Boolean program [18]. POPEYE's design consists of two distinct parts:

1. ***Dependent Type Checking.*** Initially, we infer coarse dependent types for all local expressions within a procedure using dependent type rules that encode intraprocedural path information in terms of first-order logic formulae that range over linear arithmetic and uninterpreted functions, the latter used to abstract a program's higher-order control-flow. We build verification conditions that exploit the dependent types and which are subsequently supplied into a first-order decision procedure. Verification failure yields an intraprocedural counterexample path.
2. ***Dependent Type Refinement.*** The counterexample path can be used by existing predicate discovery algorithms to appropriately strengthen pre- and post-conditions at function calls. Newly discovered refinement predicates are propagated along subtyping chains that capture interprocedural dependencies to strengthen the dependent type signatures of the procedures used at these call-sites.

The remainder of the paper is organized as follows. In the next section, we present an informal overview of our approach. Section 3 defines a small dependently-typed higher-order core language. We formalize our verification strategy for this language in Section 4. Section 5 discusses the implementation and experimental results. Related work and conclusions are given in Sections 6 and 7.

2 Overview and Preliminaries

Dependent Types. We consider two kinds of dependent type expressions:

1. a *dependent base type* written $\{\nu : B \mid r\}$, where ν is a special value variable undefined in the program whose scope is limited to r, B is a base type, such as int or bool, and r is a boolean-valued expression (called a *refinement*). For instance, $\{\nu : \mathsf{int} \mid \nu > 0\}$ defines a dependent type that represents the set of positive integers.
2. a *dependent function type* written:

$$\{x : P_{1_x} \to P_1\} \oplus \{x : P_{2_x} \to P_2\} \oplus \ldots \oplus \{x : P_{n_x} \to P_n\}$$

abbreviated as $\oplus_i\{x : P_{ix} \to P_{io}\}$, where each $\{x : P_{i_x} \to P_i\}$ defines a function type whose argument x is constrained by dependent type P_{ix} and whose result type is specified by P_i. The different components of a dependent function type distinguish different contexts in which the function may be used. For instance,

$$\{x : \{\nu : \mathsf{int} \mid \nu > 0\} \to \{\nu : \mathsf{int} \mid \nu > x\}\} \oplus \{x : \{\nu : \mathsf{int} \mid \nu < 0\} \to \{\nu : \mathsf{int} \mid \nu < x\}\}$$

specifies the function that, in one call-site, given a positive integer returns an integer greater than x, while in another, given a negative integer returns an integer less than x. Components in a dependent function type are indexed by an implicit label, e.g., a finite call-string used in polyvariant control-flow analyses [24,12].

As shorthand, we sometimes write only the refinement predicate to represent the dependent type, omitting its type constructor. Thus, in the following, we sometimes write $\{r\}$ as shorthand for $\{\nu : B \mid r\}$. For example, $\{\nu > 0\}$ represents $\{\nu : \mathsf{int} \mid \nu > 0\}$. We also write B as shorthand for $\{\nu : B \mid \mathsf{true}\}$. For perspicuity, we use syntactic sugar to allow the $\oplus$ operator to be "pushed into" refinements:

$$\{\nu : B \mid r_1\} \oplus \{\nu : B \mid r_2\} = \{\nu : B \mid r_1 \oplus r_2\}$$
$$\{x : P_1 \to P_{r_1}\} \oplus \{x : P_2 \to P_{r_2}\} = \{x : P_1 \oplus P_2 \to P_{r_1} \oplus P_{r_2}\}$$

As a result, context-sensitive dependent types reuse the shape of ML types (Section 3.1). Additionally, we define $P.i$ to return the dependent type indexed by label i. When a function is used in a single context, we simply write $\{x : P_x \to P\}$.

Procedure Specifications. A procedure specification is given in terms of a pre- and post-condition of a procedure; we express these conditions in terms of a dependent function type where the type of the function's domain can be thought of as the function's pre-condition, and where the type of the function's range defines its post-condition.

```
fun f g x =
  if x>=0 then
    let r = g x in r
  else
    let p = f g
        q = compute x
        s = f p q
    in s
```

```
fun main h n =
  let r = f h n
  in assert (r >= 0)
```

Fig. 1. The use of higher-order procedures can make compositional dependent type inference challenging

2.1 Example

Consider the program shown in Fig. 1. This program exhibits complex dataflow (e.g., it can create an arbitrary number of closures via the partial application of `f`) and makes heavy use of higher-order procedures (e.g., the formal parameter `g` in function `f`). We wish to infer a useful specification for `f` without having to (a) supply candidate qualifiers used in the dependent types that define the specification, (b) know the possible concrete arguments that can be supplied to `g`, or (c) require details about `compute`'s definition. In spite of these constraints, our technique nonetheless associates the following non-trivial type to `f`:

$$\mathtt{f} : \{\mathtt{g} : \{\mathtt{g}_{\mathtt{arg}} : \{\nu \geq 0\}\} \rightarrow \{\nu \geq 0\}\} \rightarrow \mathtt{x} : \{\mathsf{true}\} \rightarrow \{\nu \geq 0\}\}$$

This type ascribes an invariant to `g` that asserts that `g` must take a non-negative number as an argument (as a consequence of the path constraint `(x >= 0)` within which it is applied) and returns a non-negative number as a result (as a consequence of the assertion made in `main`).

```
fun g x y = x
fun twice f x y =
  let p = f x
  in f p y
fun neg x y = -(x ())
```

```
fun main n =
  if n >= 0 then
    assert(twice neg (g n) () >= 0)
  else ()
```

Fig. 2. A function's specification can be refined based on the context in which it is used

The utility of context-sensitive dependent types arises when a function is called in different (potentially inconsistent) contexts. Consider the program shown in Fig. 2. Here, function `f` (which is supplied the argument `neg` in `main`) is called in two different contexts in the procedure `twice`. The first argument to `f` is a higher-order procedure - in the first call, this procedure is bound to the result of evaluating `g n`; in the second call, the procedure (bound to `p`) is the result of the first partial application. Since `f` negates the value yielded by applying its procedure argument to `()`, we thus infer the following specification:

$$\mathtt{f}_{\mathtt{arg}_1} : \{\{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \geq 0 \ \oplus \ \nu \leq 0\}\} \rightarrow \mathtt{f}_{\mathtt{arg}_2} : \{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \leq 0 \ \oplus \ \nu \geq 0\}$$

3 Language

We formalize our ideas in the context of a call-by-value variant of the λ-calculus with support for dependent types. The syntax of the language is shown in Fig. 3. We use $f, g, x, y, \ldots$ to range over variables; typically, f and g (as well as their subscripted variants) are only bound to abstractions, while x and y (as well as their subscripted variants) can be bound to values of any type. The special variable ν is used to denote the value of a term in its corresponding dependent type refinement predicate. The language supports a small set of base types (B), monotypes (τ), type schemas (σ) that introduce polymorphic types via type variables that are universally quantified at the outermost level, and dependent types (P) that include dependent base types and dependent function types.

Predicates (p) are Boolean expressions built from a predefined set ($\mathcal{Q}$) of first-order logical, arithmetic, and relational operators; the arguments to these operators are simple expressions - variables, constants, or function applications; to simplify the technical development, we assume function applications are A-normalized, ensuring every abstraction and function argument is associated with a program variable. A refinement expression is either a refinement variable (κ) that represents an initially unknown refinement or a concrete predicate (p). Templates (P_T) are dependent types whose refinement expressions are only refinement variables (κ). The pick or selection operator $\kappa.i$ on refinement variable allows $\oplus$ to be pushed into refinements (as described in Section 2), and hence omitted in template definitions. Instantiation of the refinement variables to concrete predicates takes place through the type refinement algorithm described in Section 4. An assert statement of the form "$\mathsf{assert}\ p; e'$" evaluates expression e' if predicate p evaluates to true and returns the special value fail otherwise.

$$f, g, x, y, \ldots \in \mathsf{Var} \quad c \in \mathsf{Constant} ::= 0, 1, \ldots, \mathsf{true}, \mathsf{false}, \ldots \quad B \in \mathsf{Base} ::= \mathsf{int} \mid \mathsf{bool}$$
$$\tau \in \mathsf{Monotype} ::= B \mid \alpha \mid \tau \to \tau \qquad \sigma \in \mathsf{PolyType} ::= \tau \mid \forall \alpha.\sigma$$
$$P \in \mathsf{DepType} ::= \{\nu : B \mid r\} \mid \oplus_i \{x : P \to P\}$$
$$r \in \mathsf{Refinement} ::= \kappa \mid p \quad p \in \mathsf{Predicate} ::= \mathcal{Q}(s, \ldots, s)$$
$$\kappa \in \mathsf{RefinementVar} ::= \kappa \mid \kappa.i \quad P_T \in \mathsf{Template} ::= \{\nu : B \mid \kappa\} \mid \{x : P_T \to P_T\}$$
$$s \in \mathsf{SimpleExp} ::= \nu \mid x \mid f\ x \quad v \in \mathsf{Value} ::= c \mid \lambda x.e$$
$$e ::= s \mid v \mid \mathsf{fix}\ e \mid \mathsf{fail} \mid \mathsf{if}\ p\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{assert}\ p;\ e \mid [\Lambda\alpha]\, e \mid [\tau]\, e$$

Fig. 3. Syntax

3.1 Dependent Type System

Type inference and checking use an ordered *type environment* Γ that consists of a sequence of dependent type bindings $x : P_x$ along with guard expressions drawn from conditional expression predicates. The use of these guard expressions makes

the type system path-sensitive since the dependent types inferred for a term are computed using the guard expressions that encode the program path taken to reach this term. We define the *shape* of a dependent type as its corresponding ML type; thus, $Shape(P)$ is obtained by replacing all refinements in P with true. We generalize its definition to type environments in the obvious way - hence, $Shape(\Gamma)$ consists only of bindings that relate variables to ML types, with all refinements replaced with true and guard expressions found in Γ removed.

$$\frac{\Gamma(x) = \{\nu \ : \ B \mid e\}}{\Gamma \vdash x \ : \ \{\nu \ : \ B \mid \nu = x\}} \ \mathsf{VarBase} \qquad \frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash x : \Gamma(x)} \ \mathsf{VarFunc}$$

$$\frac{}{\Gamma \vdash c : ty(c)} \ \mathsf{Const} \qquad \frac{\Gamma \vdash e_1 : P_1 \quad \Gamma; x : P_1 \vdash e_2 \ : \ P_2}{\Gamma \vdash \mathsf{let}\ x \ = \ e_1\ \mathsf{in}\ e_2 : P_2} \ \mathsf{Let}$$

$$\frac{\forall\, i.\ \Gamma_i; x : P_{ix} \vdash e : P_{ie} \quad \Gamma_i; x : P_{ix} \vdash P_{ie} <: P_i}{\oplus_i \Gamma_i \vdash \lambda x.e : \oplus_i \ \{x : P_{ix} \to P_i\}} \ \mathsf{Fun} \qquad \frac{\Gamma \vdash \ e \ : \ \{f : P_f \to \oplus_i \ \{x : P_{ix} \to P_i\}\}}{\Gamma; f : P_f \vdash \mathsf{fix}\ e \ : \ \oplus_i \ \{x : P_{ix} \to P_i\}} \ \mathsf{Fix}$$

$$\frac{\Gamma \vdash y \ : \ P_y \quad \Gamma \vdash P_y <: P_x \quad \Gamma \vdash f_i : (x : P_x \to P)}{\Gamma \vdash f_i(y) : [y/x]P} \ \mathsf{App}$$

$$\frac{\Gamma \vdash p : \mathsf{bool} \quad \Gamma; p \vdash e_t : P_t \quad \Gamma; \neg p \vdash e_f : P_f}{\Gamma \vdash \mathsf{if}\ p\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \ : \ \mathcal{C}(p, P_t, P_f)} \ \mathsf{If} \qquad \frac{\Gamma \vdash f : \oplus_i \ \{x : P_{ix} \to P_i\}}{\Gamma \vdash f_j : \{x : P_{j_x} \to P_j\}} \ \mathsf{Pick}$$

$$\frac{\forall\, i.\ \Gamma_i \vdash f_i : \{x : P_{ix} \to P_i\}}{\oplus_i \Gamma_i \vdash f : \oplus_i \ \{x : P_{ix} \to P_i\}} \ \mathsf{Conc} \qquad \frac{\Gamma \vdash \{\mathsf{bool} \mid \mathsf{true}\} <: \{\mathsf{bool} \mid p\} \quad \Gamma \vdash e \ : \ P}{\Gamma \vdash \mathsf{assert}\ p;\ e \ : \ P} \ \mathsf{Assert}$$

$$\frac{\Gamma \vdash e : \forall \alpha.P \quad \Gamma \vdash P' \quad Shape(P') = \gamma}{\Gamma \vdash [\gamma]e : [P'/\alpha]P} \ \mathsf{Inst} \qquad \frac{\Gamma \vdash e : P \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash [\Lambda\alpha]e : \forall \alpha.P} \ \mathsf{Gen}$$

$$\frac{\langle \Gamma \rangle \wedge \langle r_1 \rangle \Rightarrow \langle r_2 \rangle}{\Gamma \vdash \{\nu \ : \ B \mid r_1\} <: \{\nu \ : \ B \mid r_2\}} \ \mathsf{BaseSub} \qquad \frac{\Gamma \vdash P'_x <: P_x \quad \Gamma; x : P'_x \vdash P <: P'}{\Gamma \vdash \{x : P_x \to P\} <: \{x : P'_x \to P'\}} \ \mathsf{FunSub}$$

$$\frac{\forall\, i.\ \Gamma \vdash \{x : P_{ix} \to P_i\} <: \{x : P'_{ix} \to P'_i\}}{\Gamma \vdash \oplus_i \ \{x : P_{ix} \to P_i\} <: \oplus_i \ \{x : P'_{ix} \to P'\}} \ \mathsf{ConcFunSub}$$

Fig. 4. Dependent typing rules

Fig. 4 defines the dependent type inference rules; these rules are adapted from [23], generalized to deal with richer path and context-sensitive types. Syntactically, $\Gamma \vdash e : P$ states that expression e has type dependent type P under type environment Γ. Our typing rules are refinements of the ML typing rules. If $\Gamma \vdash e : P$ then $Shape(\Gamma) \vdash e : Shape(P)$. $\Gamma; \mathtt{x} : P$ defines the type environment that extends the sequence Γ with a binding for $\mathtt{x}$ to P. The rules for variables, constants, let-expressions are standard. Rule Fun associates a context-sensitive dependent function type with an abstraction. The structure of this type is determined by the different contexts in which the abstraction is applied (Γ_i) generated

from rule Conc described below. The first judgment in the antecedent considers the type of the abstraction body in all type environments Γ_i enriched by a type binding of bound variable x with dependent type P_{i_x}. The second judgment asserts that P_{i_e}, the type associated with the body of the abstraction, be a subtype of the return type of the abstraction. Rule Fix defines recursive functions in the obvious way. Rule App establishes a subtyping relation between the actual and formal parameters in the application. The abstract labels that subscript function identifiers in the rules are used to express context-sensitivity but are not part of the program syntax, and are constructed during the interprocedural type refinement phase.

In the If rule, we independently infer types P_t and P_f for branch expressions e_t and e_f, resp. Then, the dependent type of the entire expression is given using operator $\mathcal{C}$ that enforces the guard expression (or its negation) p (or $\neg p$) to be a precondition of the corresponding type:

$$\mathcal{C}(p, \{\tau \mid r_1\}, \{\tau \mid r_2\}) = \{\tau \mid p \Rightarrow r_1 \wedge \ \neg p \Rightarrow r_2\}$$
$$\mathcal{C}(p, \oplus_i\{x : P_1 \to P_{r_1}\}, \oplus_i\{x : P_2 \to P_{r_2}\}) = \oplus_i\{x : \mathcal{C}(p, P_1, P_2) \to \ \mathcal{C}(p, P_{r_1}, P_{r_2})\}$$

There are two rules for extracting and generating context-sensitive dependent type functions. A term f with type $\oplus_i \ \{x : P_{i_x} \to P_i\}$ reflects the type of all uses of f in different contexts; the type at a given context can be indexed by the label at the use (rule Pick). Conversely, we can construct the concatanation of the types at each context to yield the actual type of the function (rule Conc). The subtype judgment in rule Assert enforces that the assertion predicate p hold. Polymorphic instantiation and generalization are defined in the standard way.

There are three subtyping rules. In rule Base Subtyping, the premise check $\langle\Gamma\rangle \wedge \langle r_1\rangle \Rightarrow \langle r_2\rangle$ requires that the conjunction of environment formula $\langle\Gamma\rangle$ and r_1 imply r_2. As in [23], $\langle\Gamma\rangle$ is defined as a first order logic formula:

$$\bigwedge\{r \mid r \ \in \Gamma\} \wedge \bigwedge\{[[x/\nu]\ r] \mid x \ : \ \{\nu \ : \ B \mid r\} \in \Gamma\}$$

Rule FunSubtype defines the usual subtyping relation on functions and rule ConcFun generalizes this rule to deal with context-sensitivity. These three rules implicitly encode subtyping chains, allowing specifications to be propagated across function boundaries.

Our semantics enjoys the usual progress and preservation properties; evaluation preserves types, and well-typed programs do not get stuck. (An assertion violation causes the program to halt with the special value fail.)

Theorem 1 (Dependent Type Safety)

1. *(Preservation) If* $\Gamma \vdash e : P$ *and* $e \hookrightarrow e'$ *then* $\Gamma \vdash e' : P$
2. *(Progress) If* $\Gamma \vdash e : P$*, where* $e \ \neq$ fail *then* e *is either a constant or an abstraction, or there exists an* e' *such that* $e \hookrightarrow e'$.

4 Verification Procedure

Our verification system consists of (a) a type-checking algorithm that encodes intra-procedural path constraints and generates verification conditions whose validity can be checked by a first-order decision procedure, and (b) a counterexample guided dependent type refinement loop that uses the counterexample yielded by a verification failure to strengthen existing invariants, and propagate new ones inter-procedurally via dependent subtyping chains.

4.1 Dependent Type Checking

The first step of our verification procedure is to assign each function a dependent type *template* as described earlier. By applying our inference rules, with the type template, given a type environment Γ and expression e, we can construct dependent types for local expressions and derive a set of subtyping constraints, which will be subsequently used to generate verification conditions (VC).

There are three verification conditions generated from the type checking rules. First, a subtyping constraint introduced by an assert expression:

$$\Gamma \vdash \{\mathsf{bool} \mid \mathsf{true}\} <: \{\mathsf{bool} \mid p\}$$

entails a verification condition that checks the validity of p under the path constraints and type bindings defined by Γ. Second, the subtyping constraint associated with function abstraction:

$$\Gamma; x : P_{T_x} \vdash P_{T_e} <: P_T$$

establishes a verification condition on the post-condition of this abstraction that requires it be consistent with the invariants inferred for its body. Third, the subtyping constraint associated with function application:

$$\Gamma \vdash P_{T_y} <: P_{T_x}$$

entails a verification condition that checks that the specification of the function's pre-condition subsumes the invariants associated with the argument at the call.

A solution in our system is defined by a refinement environment Σ that maps refinement variables κ to refinements. We lift this notion to dependent types $\Sigma(P_T)$ and type environment $\Sigma(\Gamma)$ by substituting each place holder κ with $\Sigma(\kappa)$ appearing in P_T and Γ. A verification condition c is valid if $\Sigma(c)$ is valid. We say Σ satisfies a subtyping constraint $\Gamma \vdash P_{T1} <: P_{T2}$ if $\Sigma(\Gamma) \vdash \Sigma(P_{T1}) <: \Sigma(P_{T2})$. Σ is a valid solution if it satisfies all subtype constraints.

Like [23], we deconstruct arbitrary subtyping constraints to base subtyping constraints (Fig. 4). According to the Base Subtyping rule, the verification condition formula is generated as

$$\langle \Sigma(\Gamma) \rangle \wedge \langle \Sigma(r_1) \rangle \Rightarrow \langle \Sigma(r_2) \rangle$$

To allow our verification engine to deal with unknown higher-order functions, we encode higher-order functions into an uninterpreted form. Suppose the type

of function f is $x_0 : P_{x_0} \rightarrow \cdots \rightarrow x_n : P_{x_n} \rightarrow P_f$. We encode P_f to be $\{\nu = \mathsf{R}_f(\mathsf{arg}_0(f), \ldots, \mathsf{arg}_n(f))\}$; here, R_f and arg_i are uninterpreted terms representing the result of function f and the i^{th} argument supplied to f at a call. Applications of higher-order function f are encoded by substituting actuals for the appropriate (suitably encoded) formal as $\mathsf{Encode}(f)$. This gives us the ability to verify a function modularly without having to know the set of definitions referenced by a functional argument or result. For example, for the program shown in Fig. 1, the variable `r` in the let-binding, `r = g x`, is encoded as $[\mathtt{x}/\mathsf{arg}_0(\mathtt{g})]\mathsf{R}_\mathtt{g}(\mathsf{arg}_0(\mathtt{g}))$, which is simply $\mathsf{R}_g(\mathtt{x})$. The subtyping constraint built for checking the post-condition during the verification of `f`, leads to the construction of the verification condition:

$$((\mathtt{x} \geq 0 \wedge \mathtt{r} = \mathsf{R}_g(\mathtt{x})) \Rightarrow \nu = \mathtt{r}) \wedge ((\neg(\mathtt{x} \geq 0) \wedge \mathtt{s} \geq 0) \Rightarrow \nu = \mathtt{s}) \Rightarrow (\nu \geq 0)$$

4.2 Dependent Type Refinement

The heart of our counterexample-guided type refinement loop is given in Fig. 5. Our refinement algorithm exploits the dependent type template and subtyping constraints generated from type inference rules and finally returns solution Σ. In `Solve`, our method iteratively type checks each procedure of the given program using the subtyping rules listed in Fig. 4 until a fix-point is reached. When a procedure cannot be typed with the set of current refinements, our method supplies the unverified procedure's type environment Γ, the current refinement map Σ, its type template $x : P_{T_x} \rightarrow P_T$, the unverified function $\lambda x.e$, and the verification conditions C constructed for the function to `Refine` which can then proceed to strengthen the function's dependent type.

Counterexample Generation. Our refinement algorithm first constructs a counterexample *ce* for an unverified verification condition. The counterexample is derived by solving the negation of the desired verification condition:

$$\langle \Sigma(\Gamma) \rangle \wedge \langle \Sigma(r_1) \rangle \wedge \neg \langle \Sigma(r_2) \rangle$$

The encoding of Γ and r_1 reflects path information; by the structure of the rules in Fig. 4, the encoding of refinement r_2, on the other hand, reflects a safety property that is implied by $\langle \Gamma \rangle \wedge \langle r_1 \rangle$. Thus, an assignment to this formula leads to a counterexample of a possible safety violation; this counterexample path is represented as a straight-line program.

A path expression of the form: "`if` p `then` e_t `else` e_f" is translated to: "`assume` p; e_t" if an assignment from the VC evaluates p to `true` and "`assume` $\neg p$; e_f" otherwise. Consider our example from Fig. 1. A first-order decision procedure would find an assignment to the the negation of the VC as an error witness, e.g., `r = -1` and `x = 1`. The representation of the counterexample path of procedure `f` given in Fig. 1 is thus:

```
fun f g x = assume (x >= 0); let r = g x in r
```

```
Refine (Γ, Σ, {x : P_{T_x} → P_T}, λx.e, C) =
  if exists c ∈ C such that Σ(c) is not valid with a witness of ce
  then
    let Σ' = case c of
      | Γ ⊢ {ν : B|p1} <: {ν : B|r2} ⇒
        let pred = case r2 of | p2 ⇒ r2 | _ ⇒ Σ[r2]
        in Strengthen ({x : P_{T_x} → P_T}, Σ, wp (ce, pred), r2)
      | Γ ⊢ {ν : B| κ1} <: {ν : B| κ2} ⇒
        Σ[κ1 ↦ (Σ[κ1]) ∧ (Σ[κ2])]
    in Refine (Γ, Σ', {x : P_{T_x} → P_T}, λx.e, C)
  else Σ

Solve (procedures as List[Γ, {x : P_{T_x} → P_T}, λx.e, C], Σ) =
  if exists (Γ, {x : P_{T_x} → P_T}, λx.e, C) for a procedure needs to be checked
  then
    Solve (procedures, Refine (Γ, Σ, {x : P_{T_x} → P_T}, λx.e, C))
  else Σ
```

Fig. 5. Type refinement algorithm

According to the two different forms of subtyping constraints generated, dependent types can be refined from the counterexample path in one of two ways: weakest precondition generation or procedure specification propagation.

Weakest Precondition Generation. In this setting, the constraint is of the form: $\Gamma \vdash \{\nu : B \mid p_1\} <: \{\nu : B \mid r_2\}$, corresponding to the first case in `Refine` in Fig. 5, where p_1 is a concrete predicate and r_2 is either a concrete predicate or a refinement variable or a selection of refinement variable. This constraint is generated when based typed expression is supplied as function argument or return or establishing assertions. Our type refinement in this case can be implemented by a backward symbolic analysis analogous to weakest precondition generation, operating over a counterexample path. Recall that the weakest pre-condition of an expression S is a function $wp(S, Q)$ mapping the post-condition Q to a pre-condition P, ensuring the execution of S terminates in a final state satisfying Q. Similarly, our weakest precondition generation simply pushes up post-conditions backwards, substituting terms for values in the presumed post-condition based on the structure of the term used to generate the pre-condition.

Our weakest precondition semantics is extended to deal with counterexample paths that include unknown function calls but for which context information constraining their arguments or results is available. Here, we can only strengthen relevant signatures, deferring the re-verification of the procedure being invoked until it becomes known. The called function's post-condition will be eventually propagated via dependent subtyping chains back to the procedures that flow into this call-site; in doing so, pre-conditions of these functions could be strengthened, requiring re-verification of the calling contexts in which they occur to ensure that these contexts imply the pre-condition. Such flows are handled directly by the subtyping chains analyzed by the refinement phase. For a called higher-order

$$
\begin{aligned}
\mathsf{wp}(e, \phi) = \ & \mathsf{case}\ e\ \mathsf{of} \\
& \mid \lambda x.e \Rightarrow \mathsf{wp}(e, \phi) \\
& \mid \mathtt{assume}\ \psi;\ e \Rightarrow (\psi \Rightarrow \mathsf{wp}(e, \phi)) \\
& \mid \mathtt{let}\ x\ =\ e'\ \mathsf{in}\ e \Rightarrow \mathsf{wp}(x\ =\ e',\ \mathsf{wp}(e, \phi)) \\
& \mid v = x \Rightarrow [x/v]\phi \\
& \mid v = c \Rightarrow [c/v]\phi \\
& \mid v = f\,\overrightarrow{d} \Rightarrow [\mathsf{Encode}(f)/v]\phi \\
& \mid \_ \Rightarrow \mathsf{wp}(\nu\ =\ e, \phi)
\end{aligned}
$$

Fig. 6. Weakest precondition generation definition

function f, we use $\mathsf{Encode}\ (f)$ to represent its value. The definition of our wp is given in Fig. 6. Consider the example in Fig. 1. The post-condition inferred is $\nu \geq 0$. We can infer the precondition shown in Section 2 by applying our wp rules as follows:

$$
\begin{aligned}
& \mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0); \mathsf{let}\ \mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\ \mathsf{in}\ \mathtt{r}\,), \nu \geq 0) = \\
& \mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0), \mathsf{wp}(\mathsf{let}\ \mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\ \mathsf{in}\ \mathtt{r}\,, \nu \geq 0)) = \\
& \mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0), \mathsf{wp}(\,\mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\,, (\mathsf{wp}(\nu = \mathtt{r}\,, \nu \geq 0)))) = \\
& \mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0), \mathsf{wp}(\,\mathtt{r}\ =\ \mathtt{g}\ \ \mathtt{x}\,, \mathtt{r} \geq 0)) = \\
& \mathsf{wp}(\,\mathtt{assume}\,(\mathtt{x} \geq 0), \mathsf{R}_g(\mathtt{x}) \geq 0) = \\
& \mathtt{x} \geq 0 \Rightarrow \mathsf{R}_g(\mathtt{x}) \geq 0
\end{aligned}
$$

Thus g's specification is strengthened to $\mathtt{g} : \{\{\nu \geq 0\} \rightarrow \{\nu \geq 0\}\}$.

When a function call `f(x)` is encountered and the abstraction to which `f` is bound is known precisely (e.g., based on a syntactic or control-flow analysis preprocessing phase), our method strengthens the post-condition of the function's body of `f` to that available at the call. `wp` recursively applies our verification technique to refine the function's precondition based on the post-condition defined by the context in which it is called. `wp` can then be executed from this call site operating on the rest of statements of the counterexample beyond the call site and the newly strengthened precondition.

Procedure Specification Propagation. In this setting, the subtyping constraint is of the form: $\Gamma \vdash \{\nu\ :\ B \mid \kappa_1\} <: \{\nu\ :\ B \mid \kappa_2\}$, corresponding to the second case in `Refine` in Fig. 5. Refinement variables are introduced when defining dependent type templates; this occurs during inference of function abstraction and fix expressions. Ensuring the subtyping constraint holds requires that any instantiation of κ_2 be propagated to κ_1. This enables refinements associated with the post-condition of a higher order function to be propagated into the real function body, and conversely to propagate refinements associated with a function's pre-condition back to the parameters of higher order function.

Consider how we might verify the program shown in Fig. 2. Our method initially infers a dependent type template for `f` as $\{\{\kappa_{1_1} \rightarrow \kappa_{1_2}\} \rightarrow \kappa_2 \rightarrow \kappa_{\mathtt{f}}\}$. The assertion in `main` drives a new post-condition $\{\nu \geq 0\}$ for `twice`, and hence $\mathtt{f}_2$ which is the second the call to `f`, instantiating $\kappa_{\mathtt{f}}$ to $\{\mathsf{true} \oplus \nu \geq 0\}$. This constraint is then propagated to the post-condition of `neg` since `neg` subtypes to `f` at the call site of `twice` in `main`. The weakest pre-condition backward analysis of our system then strengthens the pre-condition for `neg` and propagates it back to `f`, instantiating $\{\kappa_{1_2}\}$ to $\{\mathsf{true} \oplus \nu \leq 0\}$. In `twice`, our technique needs to ensure, at the second call site of $\mathtt{f}_2$, the actual higher-order function `p` subtypes to the first argument of `f` where `p` is derived from the first call to `f` notated as $\mathtt{f}_1$. The subtyping relation can then be expressed as $\Gamma \vdash \{\kappa_2.1 \rightarrow \kappa_{\mathtt{f}}.1\} <: \{\kappa_{1_1}.2 \rightarrow \kappa_{1_2}.2\}$. The post-condition in $\kappa_{1_2}.2$ ($\{\nu \leq 0\}$) is then propagated to $\kappa_{\mathtt{f}}.1$, which becomes $\{\nu \leq 0 \oplus \nu \geq 0\}$. Finally, the context-sensitive type for `f` is derived as

$$\mathtt{f}_{\mathtt{arg}_1} : \{\{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \geq 0 \ \oplus \ \nu \leq 0\}\} \rightarrow \mathtt{f}_{\mathtt{arg}_2} : \{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \leq 0 \ \oplus \ \nu \geq 0\}$$

4.3 Correctness

We provide two correctness results for our verification algorithm $\mathcal{V}(\Gamma, Prog)$ where Γ is top-level typing environment and *Prog* is a program[1]. The first (*Soundness*) states that the dependent types inferred by our verification procedure are consistent with our type rules. The second (*Weak*) states that our procedure generates the least type necessary to discharge the subtyping constraints collected by the inference algorithm. In the following, $R(\Gamma)$ recursively extracts dependent base types $\{\nu : B|\kappa\}$ from the domain of Γ.

Theorem 2 (Verification Algorithm)

1. *(Soundness) Let* $((\ldots, \{\Gamma, x : P_{T_x} \rightarrow P_T, \lambda x.e, C\}, \ldots), \Sigma)$ *be the result of* $\mathcal{V}(\Gamma, Prog)$*. Then, provided* $\mathcal{V}(\Gamma, Prog)$ *terminates,* $\Sigma(\Gamma) \vdash \lambda x.e : \{x : \Sigma(P_{T_x}) \rightarrow \Sigma(P_T)\}$.
2. *(Weak) And, for all other valid solution* Σ'*, the algorithm generates the weakest solution:* $\forall c$ *as* $\{\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}\} \in C$*, and* $\forall \{\nu : B|\kappa\} \in \{R(\Gamma) \cup \{\nu : B|r_1\}\}$, $\Sigma'(\Gamma) \vdash \{\nu : B|\Sigma'(\kappa)\} <: \{\nu : B|\Sigma(\kappa)\}$.

4.4 Invariant Generation

Because our technique does not guarantee termination given the undecidability of automatically synthesizing loop invariants, the size of a dependent function type may grow into an infinite representation, and a fixed-point may never be reached. Consider the ML program fragment shown in Fig. 7 adapted from [13]. The procedure `iteri` visits the elements of a list `xs`, applying function `f` to each element and its index in the list. Procedure `mask` calls `iteri` when the length of its array and list arguments are the same. It supplies function `g` as the

[1] The proof can be found in www.cs.purdue.edu/homes/zhu103/pubs/vmcai13full.pdf

higher-order argument to `iteri` which performs some computation involving a list and array element at the same index. We desire to verify the array bound safety property $\texttt{j} < \mathsf{len}(\texttt{a})$ for the array access in procedure g (Note $\texttt{j} \geq 0$ can be directly proved by our method introduced in Section 4.2).

```
fun iteri i xs f =
  case xs of
    [ ] => ()
  | x :: xs' =>  (f i x; iteri (i+1) xs' f)

fun mask a xs =
  let g j y = ··· y ··· Array.sub (a, j) ··· in
    if Array.length a = List.length xs then
      iteri 0 xs g
    else () end
```

Fig. 7. A program that has a non-trivial loop invariant

During the course of verifying this program, we would need to discharge a specification that forms a pre-condition for `iteri` asserting that $\mathsf{len}(\texttt{xs}) \neq 0 \Rightarrow \texttt{i} < \mathsf{len}(\texttt{a})$. However, verifying this specification requires a theorem prover to conclude that $\mathsf{len}(\texttt{xs})\text{-}1 \neq 0 \Rightarrow \texttt{i}+1 < \mathsf{len}(\texttt{a})$ as precondition for the recursive call to `iteri (i+1) xs'`. In trying to discover a counter-example to this claim, a theorem prover would likely generate an infinite number of pre-conditions, $\mathsf{len}(\texttt{xs}) - k \neq 0 \Rightarrow \texttt{i} + k < \mathsf{len}(\texttt{a})$ where $k = 0, 1, 2 \cdots$ What is required is a sufficiently strong invariant that can be used to validate the required safety properties. While programmers could certainly write such specifications if necessary, we follow the idea of interpolation-based model-checking [21] to automatically infer them when possible.

When our mainline verification algorithm diverges or reaches a pre-determined timebound during the analysis of a recursive procedure, it is unrolled incrementally together with its calling context. Our method then infers dependent type templates and generates subtyping constraints for the k-unrolled procedures. Pre-conditions of the higher order functions used in recursive procedure are propagated via subtyping chain from that of the real function they represent for. Post-conditions of the higher order functions are also propagated from that of the real function which can be obtained from our type inference algorithm. We then exploit a technique described in [27] to infer dependent types from the collected base subyping constraints. The basic idea is to use the interpolation of the first-order formulas derived from the subtyping constraints to deduce an instantiation for a given type refinement variable κ. We desire that the prover returns a more suitable refinement beyond that yielded by a weakest precondition generator. Refinements synthesized from k-unrolled non-recursive procedures are folded back to the original procedure as candidates.

For example, suppose our method discovers that it must unroll the recursive procedure `iteri` two times, obtaining the program shown below:

```
fun iteri0 i0 xs0 f0 =
  case xs0 of
    [ ] => ()
  | x0 :: xs0' =>  (f0 i0 x0; iteri1 (i0+1) xs0' f0)

fun iteri1 i1 xs1 f1 =
  case xs1 of
    [ ] => ()
  | x1 :: xs1' =>  (f1 i1 x1; iteri2 (i1+1) xs1' f1)

fun iteri2 i2 xs2 f2 = halt
```

Fig. 8. Unrolling a recursive procedure to enable loop invariant discovery using interpolation

Here, `halt` is a special term, representing a termination point. Because we maintain the original calling context of `iteri`, we have $\mathsf{len}\ \mathtt{a} = \mathsf{len}\ \mathtt{xs}$ in the typing environment and leverage subtyping constraints to establish that the actual `g` subtypes to the formal `f0`. We infer refinements for this unrolled excerpt using the obtained base subtyping constraints. We thus have the following subtyping constraint:

$$\mathtt{i1} : \kappa_{\mathtt{i1}},\ \mathtt{i0} : \kappa_{\mathtt{i0}},\ \mathtt{xs0} : \kappa_{\mathtt{xs0}}, \mathsf{len}\,(\mathtt{xs0}) = \mathsf{len}\,(\mathtt{xs0}') + 1, \mathsf{len}\,(\mathtt{xs}) = \mathsf{len}\,(\mathtt{a})$$
$$\vdash \{\nu = \mathtt{xs0}'\} <: \kappa_{\mathtt{xs1}}$$

that establishes that the actual `xs0'` given to `iteri1` subtypes to the formal `xs1`. In the body of `iteri1`, there is another constraint for the call to `f1 i1`:

$$\mathtt{i1} : \kappa_{\mathtt{i1}},\ \mathtt{xs1} : \kappa_{\mathtt{xs1}}, \mathsf{len}\,(\mathtt{xs1}) = \mathsf{len}\,(\mathtt{xs1}') + 1 \vdash \{\nu' = i1\} <: \{\nu' < \mathsf{len}\,(\mathtt{a})\}$$

Because we have already inferred the dependent type for procedure `g` before typing `iteri` and obtained precondition $\nu' < \mathtt{len(a)}$ for its first argument, we can use it to also serve as the precondition of the first argument of `f1` propagated through the subtyping chains.

We extend the above constraints into first order logic formulas:

$$\{\mathtt{i1} = \mathtt{i0} + 1 \ \wedge\ \mathtt{i0} = 0 \ \wedge\ \mathtt{xs0} = \mathtt{xs} \ \wedge\ \mathsf{len}\,(\mathtt{xs0}) = \mathsf{len}\,(\mathtt{xs0}') + 1 \wedge$$
$$\mathsf{len}\,(\mathtt{xs}) = \mathsf{len}\,(\mathtt{a}) \ \wedge\ \nu = \mathtt{xs0}'\}^{(a)} \Rightarrow \kappa_{xs1}$$
$$\kappa_{xs1} \Rightarrow \{\mathtt{i1} = \mathtt{i0} + 1 \ \wedge\ \nu = \mathtt{xs1} \ \wedge\ \mathsf{len}\,(\mathtt{xs1}) = \mathsf{len}\,(\mathtt{xs1}') + 1 \wedge$$
$$\nu' = \mathtt{i1} \Rightarrow \nu' < \mathsf{len}\,(\mathtt{a})\}^{(b)}$$

The unknown refinement represented by $\kappa_{\mathtt{xs1}}$ is indeed an interpolation of formula (a) and formula (b) and can be inferred by feeding them into an appropriate

interpolation theorem prover [21] which may return $\mathsf{len}(\nu) + \texttt{i1} = \mathsf{len}\,\texttt{(a)}$ as result. Our method then yields $\mathsf{len}(\nu) + \texttt{i} = \mathsf{len}\,\texttt{(a)}$ (discarding subscript) as a refinement candidate of the second argument `xs` of procedure `iteri`.

After candidate refinement synthesis, our method then applies an elimination procedure [23] to filter out incorrect candidates. If the original procedure is still not typable, the process is repeated, unrolling it $k+1$ times. For this example, with the above refinement candidate, we can correctly verify the pre-condition of `f` in `iteri`. Since the theorem prover can use the case condition to know $\mathsf{length}(\texttt{xs}) > 0$ and based on the invariant $\texttt{i} + \mathsf{len}(\texttt{xs}) = \mathsf{len}(\texttt{a})$, it can determine that $\texttt{i} < \mathsf{len}(\texttt{a})$ must hold. Our method finally generates the appropriate dependent type for `iteri` as:

$$\begin{aligned}\texttt{iteri} : \texttt{i} : \mathsf{int} \rightarrow \{\texttt{xs} \ : \ '\texttt{a}\ \mathsf{list}|\ \texttt{i} + \mathsf{len}(\nu) = \mathsf{len}(\texttt{a})\} \rightarrow \\ \{\texttt{f} \ : \ \{\texttt{f}_{\mathsf{arg}_1} \ : \ \mathsf{int} \mid 0 \leq \nu < \mathsf{len}(\texttt{a})\} \rightarrow '\texttt{a} \rightarrow \mathsf{unit}\} \rightarrow \mathsf{unit}\end{aligned}$$

Note the invariant generation module is only invoked when our system diverges during the verification of a recursive procedure. We differ from [27] in two respects: first, [27] does not use an elimination procedure since it tries to infer dependent types for the original program using a whole program analysis; second, we only infer refinement candidates for a non-recursive unrolled code fragment instantiated upon divergence, instead of the original whole program, greatly reducing the number of instances where interpolation computation is required.

5 Implementation

We have implemented our verification system in Popeye. Popeye takes as input an SML program (not necessarily closed) and outputs specifications inferred for the procedures defined by the program. We have provided specifications for built-in primitive datatypes as well as arrays, lists, tuples, and records that are used to bootstrap the inference procedure. The Yices theorem prover is used as the verification engine. CSIsat [5] is employed to generate interpolations when inferring candidate refinements for recursive procedures and loops. The implementation is incorporated within the MLton whole-program optimizing compiler toolchain and consists of roughly 14KLOC written in SML[2].

5.1 Case Study: Bit Vectors

To gauge Popeye's utility, we applied it to an open-source bit vector library (bitv) [6] (version 0.6). A bit vector is represented as a record of two fields, `bits`, an array containing vector's elements, and `length`, an integer that represents the number of bits that the vector holds. Operations on bit vectors should enforce the invariant that $\texttt{(bits.length - 1)} \cdot \texttt{b} < \texttt{bits.length} \cdot \texttt{b}$, where `b` is a constant that defines the number of bits intended to be stored per array element.

[2] The Popeye implementation is available at `http://code.google.com/p/popeye-type-checker/`

This invariant is assumed for all procedures. POPEYE successfully type checks the program combined with 5 manually generated preconditions (for recursive procedures as prover [5] cannot deal with mod operation heavily used in the library) by relatively longer verification time than that of DSOLVE [23] in this benchmark; however DSOLVE requires manual addition of extra 14 user-supplied qualifiers.

Bug Detection. Without any programmer annotations, POPEYE discovered an array out-of-bounds error that occurs in the `blit` function:

```
fun blit {bits=b1, length=l1} {bits=b2, length=l2}
         ofs1 ofs2 n =
  if n < 0 || ofs1 < 0 || ofs1 + n > l1
           || ofs2 < 0 || ofs2 + n > l2
  then assert false
  else unsafe_blit b1 ofs1 b2 ofs2 n
```

This function calls `unsafe_blit` only if a guard condition that checks that all offset value and the number of bits (`n`) to be copied are positive, and that the range of the copy fit within the bounds of the source and target vectors. The counterexample reported for blit procedure corresponds to an input as {length (b1)=2, length (b2)=0, l1=60, ofs1=32, l2=0, ofs2=0, len=0}. The guard holds under this assignment, but because `unsafe_blit` attempts to access the offset in the target bit-vector that is the starting point for the copy, before initiating the copy loop, an array out-of-bounds exception gets thrown. In this example, POPEYE reports a test case that serves as a witness to the bug, and can help direct the programmer to identify the source of the error. The primary novelty of this technique in this regard is its ability to generate a precise counterexample path with concrete inputs that serve as a witness to the violation without requiring explicit user confirmation as DSOLVE.

Complex Refinement Generation. Procedure `unsafe_blit` found in this library tries to copy `n` bits starting at offset `ofs1` from bit-vector `v1` to bit-vector `v2` with target offset `ofs2`. POPEYE discovers the following precondition:

$$((\mathsf{ofs2} + \mathsf{n}) - 1)/\mathsf{b}) < \mathsf{v2.length}$$

This is a non-trivial specification comprised of refinements that we believe would be difficult, in general, for programmers to construct. Systems such as DSOLVE require users to provide these qualifiers explicitly. The ability to generate non-trivial refinements automatically only using counterexamples is an important distinguishing feature of our approach compared to e.g., Liquid Types.

5.2 Experimental Results

To test its accuracy, we have applied POPEYE to a number of synthetic SML programs from the benchmark suite used to evaluate MOCHI [18]. While these

benchmarks are small (typically less than 100 LOC), they exercise complex control- and dataflow, and exploit higher-order procedures heavily, in ways intended to make dependent type inference challenging. Details of these benchmarks are provided in [18]. In the table, column num_ref denotes the number of refinements discovered by POPEYE. num_cegar shows how many iterations of the refinement loop were necessary for POPEYE to converge. prover_call gives the number of theorem prover calls; there are typically more prover calls than CEGAR loop iterations because the results of a counterexample usually entails propagation of newly discovered invariants to other contexts, thus requiring reverification (and hence additional theorem prover calls). cegar_time shows the time spent on refinement loops. run_time gives the total running time taken.

The first seven benchmarks shown in Table 1 cannot be verified by DSOLVE using its default set of simple qualifiers since either context-sensitive dependent types or non-trivial invariants are required. The last two of these seven (suffixed with `-e`) are buggy, and thus cannot cannot also be automatically proved by DSOLVE. The last two benchmarks requires recursive procedure invariants which can be synthesized by our invariant generation module. Here, a single unrolling of the recursive procedure in repeat-e was sufficient to witness the error; in contrast, POPEYE required three unrollings of the recursive procedure in array-init to find a suitable set of candidate refinements. We note that MOCHI fails to verify the `array-init` program. While MOCHI can also verify the first eight benchmarks in this table, its formulation is a bit more complex than ours, and does not easily generalize to deal with data structures and user-level datatypes.

6 Related Work

There has been much work on the use of dependent types for checking complex safety properties of ML programs. Freeman and Pfenning [10] describe a refinement type inference scheme defined in terms of an abstract interpretation over a programmer-specified lattice of refinements for each ML type, and a restricted use of intersection types to combine these refinements that still preserves decidability of type inference. DML [28] is a conservative extension of ML's type system that supports type checking of programmer-specified dependent types; the system supports a form of partial type inference whose solution depends upon the set of refinements found in a linear constraint domain.

To reduce the annotation burden imposed by systems like DML, Liquid Types [23,14] requires programmers to only specify simple candidate qualifiers from which more complex dependent types defined as conjunctions of these refinements are inferred by a whole program abstract interpretation. Our approach differs from liquid types in four important respects: (1) we attempt to infer refinements, (2) a counterexample path together with a test case can be reported as a program bug witness; (3) the type refinement fixpoint loop enables compositional verification, propagating specifications via dependent subtyping chains on demand; (4) the dependent types we inferred are context-sensitive.

Broadly related to our goals, HMC [13] also borrows techniques from imperative program verification to verify functional programs. It does so by reducing

Table 1. Benchmark Results

Program	num_ref	num_cegar	prover_call	cegar_time	run_time
fhnhn	3	4	35	0s	0.014s
neg	15	20	230	0.004s	0.18s
max	10	11	175	0.005s	0.95s
r-file	11	21	205	0.012s	1.56s
r-lock	10	18	108	0.006s	0.60s
r-lock-e	13	18	113	0.01s	0.68s
repeat-e	39	18	237	0.11s	4.87s
list-zip	2	4	149	0.01s	1.55s
array-init	35	106	3617	0.03	102.3s

the problem of checking the satisfiability of the constraints generated in a liquid type system to a safety checking problem of a simple imperative program. However, the translated imperative program loses the structure of the original source semantics. Thus, it is not obvious how we might convert a counterexample reported in the translated program into the original source for debugging.

Terauchi [26] also proposes a counterexample-guided dependent type inference scheme, albeit based on a whole-program analysis. A counterexample in his approach is an "unwound" slice of the program that is untypable using the current set of candidate types, rather than a counterexample path. Since the unfolded program may be involved in multiple program paths, many of which may not be relevant to the verification obligation, it would appear that the size of the constraint sets that needs to be solved may become quite large.

There has been much recent interest in using higher-order recursion schemes [17,19] to define expressive model-checkers for functional programs. In [18,22], predicate abstraction is proposed to abstract higher-order program with infinite domains like integers to a finite data domain; the development in these papers is limited to pure functional programs without support for data structures. Model checking arbitrary μ-calculus properties of finite data programs with higher order functions and recursions can be reduced to model checking for higher-order recursion schemes [17]. Finding suitable refinements relies on a similar constraint solving to [26,27] for a straight-line higher-order counterexample program. Such techniques involve substantial re-engineering of first-order imperative verification tools to adapt them for a higher-order setting.

One important motivation for our work is to reuse well-studied imperative program verification techniques. For example, predicate abstraction [11] has been effectively harnessed by tools such as SLAM [2] and BLAST [4] to verify complex properties of imperative programs with intricate shape and aliasing properties. Software verification tools, such as Boogie [3], ESC/Java [9] and CALYSTO [1] construct first order logic formula to encode a program's control flow. If a verification condition, expressed via programmer-specified assertions or specifications, cannot be discharged, the counterexample path can be used to refine and strengthen it.

7 Conclusion

In this paper, we present a compositional inter-procedural verification technique for functional programs. We use dependent type checking rules to generate dependent type templates for local expressions inside a procedure. Dependent subtyping rules are then used to generate verification conditions. From an unprovable verification condition, we can construct a counterexample path to infer dependent types for procedure arguments and results, and to propagate inferred specifications between procedures and call-sites where they are applied. Thus, our technique effectively leverages a variety of strategies used in the verification of first-order imperative programs within a higher-order setting.

Acknowledgements. We thank Ranjit Jhala, Aditya Nori, and Francesco Zappa-Nardelli for many useful comments and discussions. This work was supported in part by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

1. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)
2. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static Driver Verification with Under 4% False Alarms. In: FMCAD, pp. 35–42 (2010)
3. 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)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast: Applications to Software Engineering. Int. J. Softw. Tools Technol. Transf. 9, 505–525 (2007)
5. Beyer, D., Zufferey, D., Majumdar, R.: CSISAT: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
6. http://www.lri.fr/~filliatr/software.en.html
7. 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)
8. Damas, L., Milner, R.: Principal Type-Schemes for Functional Programs. In: POPL, pp. 207–212 (1982)
9. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245 (2002)
10. Freeman, T., Pfenning, F.: Refinement Types for ML. In: PLDI, pp. 268–277 (1991)
11. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Jagannathan, S., Weeks, S.: A Unified Treatment of Flow Analysis in Higher-Order Languages. In: POPL, pp. 393–407 (1995)

13. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying Functional Programs Using Abstract Interpreters. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 470–485. Springer, Heidelberg (2011)
14. Kawaguci, M., Rondon, P., Jhala, R.: Type-based Data Structure Verification. In: PLDI, pp. 304–315 (2009)
15. Knowles, K., Flanagan, C.: Type Reconstruction for General Refinement Types. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 505–519. Springer, Heidelberg (2007)
16. Kobayashi, N.: Model-Checking Higher-Order Functions. In: PPDP, pp. 25–36 (2009)
17. Kobayashi, N.: Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs. In: POPL, pp. 416–428 (2009)
18. Kobayashi, N., Sato, R., Unno, H.: Predicate Abstraction and CEGAR for Higher-Order Model Checking. In: PLDI, pp. 222–233 (2011)
19. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order Multi-Parameter Tree Transducers and Recursion Schemes for Program Verification. In: POPL, pp. 495–508 (2010)
20. Martin-Löf, P.: Constructive Mathematics and Computer Programming (312), 501–518 (1984)
21. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
22. Ong, C.H.L., Ramsay, S.J.: Verifying Higher-Order Functional Programs with Pattern-Matching Algebraic Data Types. In: POPL, pp. 587–598 (2011)
23. Rondon, P., Kawaguci, M., Jhala, R.: Liquid Types. In: PLDI, pp. 159–169 (2008)
24. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis. In: Program Flow Analysis (1981)
25. Shivers, O.: Control-Flow analysis in Scheme. In: PLDI, pp. 164–174 (1988)
26. Terauchi, T.: Dependent types from Counterexamples. In: POPL, pp. 119–130 (2010)
27. Unno, H., Kobayashi, N.: Dependent Type Inference with Interpolants. In: PPDP, pp. 277–288 (2009)
28. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: POPL, pp. 214–227 (1999)

Abstract Read Permissions: Fractional Permissions without the Fractions

Stefan Heule[1], K. Rustan M. Leino[2],
Peter Müller[1], and Alexander J. Summers[1]

[1] ETH Zurich, Switzerland
stheule@ethz.ch,
{peter.mueller,alexander.summers}@inf.ethz.ch
[2] Microsoft Research, USA
leino@microsoft.com

Abstract. Fractional Permissions are a popular approach to reasoning about programs that use shared-memory concurrency, because they provide a way of proving data race freedom while permitting concurrent read access. However, specification using fractional permissions typically requires the user to pick concrete mathematical values for partial permissions, making specifications overly low-level, tedious to write, and harder to adapt and re-use. This paper introduces *abstract read permissions*: a flexible and expressive specification methodology that supports fractional permissions while allowing the user to work at the abstract level of read and write permissions. The methodology is flexible, modular, and sound. It has been implemented in the verification tool Chalice.

1 Introduction

An important part of reasoning about concurrent programs concerns their patterns of access to memory and other shared resources. A useful aid in the specification of such patterns is to use a model of resource *permissions* that can be transferred between program entities to specify how individual threads are currently allowed to access shared resources. By allowing permissions to be fractional [4], it is possible to distinguish between acceptable read and write accesses, which is necessary for expressive reasoning about programs with shared-memory concurrency. Permissions are a fictional notion used for static reasoning about a program; they are used in specifications and during program verification, but are not present at program execution.

The traditional model of fractional permissions associates an access permission with every memory location. The permission can be divided into fractions, which can be held by and transferred between threads and method activation records. An activation record may read a memory location only if it holds a non-zero fraction of the memory location's permission. To write to the memory location, the activation record must hold the entire permission. Since fractional permissions can only be divided and combined, but never forged or duplicated,

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 315–334, 2013.

this discipline ensures race freedom for memory updates, while allowing concurrent reads. In the verification of both sequential and concurrent programs, permissions also enable *framing*; as long as the caller of a method holds on to a non-zero fraction of the permission for a memory location, it may soundly assume that the call will not affect the value stored in the location, because the callee method cannot obtain the full (write) permission to modify it.

While fractional permissions give rise to a flexible model for reasoning, writing specifications in this model can be tedious and overly low-level due to the need to work with some concrete mathematical representation of the permissions. The problem is exacerbated by the fact that programmers are concerned with permissions only in the abstract sense of reading or writing to locations; the concrete values representing these permissions are largely irrelevant.

In this paper, we present *abstract read permissions*; a novel specification methodology that allows the programmer to reason at the level of read and write permissions. Our methodology is expressive, modular, and sound. We present it in the context of implicit dynamic frames [20], but it also applies to other permission logics; in particular, separation logic [19]. This paper builds on our previous workshop presentation [8]. Our methodology is implemented in two verifiers for the class-based, concurrent language Chalice [13]; one based on verification condition generation [12] and one on symbolic execution [11].

Outline. The next section motivates abstract read permissions through an example. Sec. 3 summarizes the background concerning permission-based verification that is used in the rest of the paper. We present the core ideas of abstract read permissions in Sec. 4, and the syntax of permission expressions in Sec. 5. Sec. 6 explains the encoding of permissions in the program verifier, and Sec. 7 provides an informal soundness argument. Sec. 8 discusses issues and solution approaches for the use of abstract read permissions in monitor invariants. We discuss related work in Sec. 9 and conclude in Sec. 10.

2 Motivation

To understand the problem of writing specifications with fractional permissions, consider a class `Expr` for arithmetic expressions with a method `eval(s: State)`, which evaluates an expression in the state `s`. The method reads the state's mapping of variables to values, denoted by `s.map`. The precondition of `eval` requires read permission to `s.map`, written **`acc`**`(s.map, `π`)` for some non-zero fraction π. Here and throughout, we use a Chalice-like syntax, but there are other notations in use; for example, in separation logic [17,3,18] this condition is written as `s.map` $\overset{\pi}{\mapsto}$ _. The postcondition of the method ensures that this permission is transferred back to the caller, allowing it to re-assemble a full permission and to update the state. The resulting specification is displayed in Listing 1. The problem that we address is how to specify the permission amount π. To illustrate the challenge, we present three existing options and discuss their shortcomings.

```
method eval(s: State)
  requires acc(s.map, π) && s.map ≠ null
  ensures acc(s.map, π)
```

Listing 1. Specification of method `eval`. The method also requires read permission to the fields of its receiver, but we ignore this aspect here for brevity.

```
class Add extends Expr {
  var left, right: Expr

  method eval(s: State) {
    leftVal := call left.eval(s)
    rightVal := call right.eval(s)
    return leftVal + rightVal
  }
}
```

Listing 2. An implementation of method `Add.eval`. The specification of `eval` is presented in Listing 1.

Concrete Fractions. One option is to represent π as a *concrete fraction*, such as `.18`. This approach has four major shortcomings:

(1) *Re-usability*: Using a concrete fraction forbids calls from a context in which the caller has a smaller fractional permission to `s.map`, say `.17`. If the specifications cannot be changed to use different fractions (for instance, when `eval` and its caller are library methods), the call is not permitted even though the caller holds a read permission, and `eval` needs read permission.

(2) *Abstraction*: For many implementations, the permission amount is irrelevant so long as it provides read access. Nevertheless, this approach forces programmers to choose a specific value and, thus, to clutter up the specification.

(3) *Framing*: Consider a subclass `Add` of `Expr` that represents the addition of two sub-expressions. Its implementation of `eval` recursively evaluates the left and right operands, and returns the sum of the results, as shown in Listing 2.

The recursive call `left.eval(s)` consumes the entire permission amount that the caller has (here, `.18`) before returning the permission to the caller. Therefore, modular verification of `Add.eval` must make the worst-case assumption that the callee (which is dynamically bound to a statically-unknown implementation) might obtain full permission to `s.map` and modify the field. Thus, the first recursive call removes any information that the caller has about the value of `s.map` before the call: in particular, that the field has a non-null value. Verification of the second recursive call then fails, since the second conjunct of the precondition (i.e., `s.map` $\neq$ `null`) cannot be proved. Failing to frame this information even though the calls require only read permission weakens verification considerably.

(4) *Permission Splitting*: Since the recursive calls to `eval` only read `s.map`, it seems reasonable that they could be called in parallel, as shown in Listing 3.

```
leftTk := fork left.eval(s)
rightTk := fork right.eval(s)
return (join leftTk) + (join rightTk)
```

Listing 3. A parallel implementation of method `Add.eval`. In Chalice, the **fork** statement launches an asynchronous method call, which entails checking the method's precondition and transferring the specified permissions to the new thread. The local variables `leftTk` and `rightTk` store thread identifiers (*tokens*) that are used in the **join** statements to join the threads and to obtain the results of the forked method calls, as well as the permissions they return.

However, there is no concrete fraction that one could choose for π that lets one verify this implementation. The entire permission of the forking thread is transferred to a new thread during the first fork, meaning that no permission is left to satisfy the precondition of the second fork.

Counting Permissions. A second option is to use *counting permissions* [3], which allow one to split a write permission into any positive number of *indivisible* units. Such a unit then grants read access. So we could represent π in the specification of `eval` as a number of such units; say one unit.

Counting permissions address the first two shortcomings of concrete fractions, but not shortcomings (3) and (4): as with concrete fractions, the recursive call `left.eval` in Listing 2 consumes all of the permission to `s.map` that the caller has and, thus, doesn't permit the framing of any information about its value; the second call then does not verify for the same reason as before. Moreover, a specification using a fixed number of units does not permit the parallel implementation in Listing 3 since, again, the first fork consumes the entire permission of the forking thread such that no permission remains to satisfy the precondition of the second fork. The number of units used in a method pre-condition also imposes a static bound on the amount of parallelization possible in a method implementation, breaking abstraction and impairing re-use.

Ghost Parameters. A third option is to let the calling context decide which permission amount to transfer to a method call; that is, what π should be for a particular call. This can be achieved by making π a parameter of the method. The parameter can be considered a *ghost* parameter, since it is needed only for the proof and can be omitted in the executing program. In separation logic, such a parameter π is typically represented as a logical variable that is bound for each invocation of the method. Both approaches are very similar: ghost parameters require programmers to provide a value for the ghost parameter when implementing a call, whereas logical variables require the verifier to provide a value when reasoning about a call. The specification for method `eval` using a ghost parameter is shown in Listing 4.

When this option is applied to *all* methods that require read permission, it addresses many of the shortcomings described above. It solves the problems of

```
method eval(s: State, ghost π: rational)
  requires 0 < π && π ≤ 1 && acc(s.map, π)
  ensures acc(s.map, π)
```

Listing 4. Specification of method `eval` using a ghost parameter π

re-use, framing, and permission splitting by allowing a caller to choose a value for the ghost parameter that ensures that it has enough permission to make the desired calls (even in parallel), and that some permission remains, to enable framing. The abstraction problem is reduced, but specifications are still cluttered up with ghost parameters (or logical variables) and constraints over them.

The abstract read permissions presented in this paper provide most of the flexibility and expressiveness of the ghost-parameter option above, but without requiring the user to manually specify the π values or constraints over them. In particular, our specification methodology does not require a concrete mathematical representation of permission amounts; especially no concrete fractions.

3 Background

Abstract read permissions are independent of any particular permission logic or verification technique. For concreteness, we present them for the Chalice language and verifier [12], which is based on the verification condition generator Boogie [1]. This section introduces the background on Chalice that is needed in the remainder of the paper.

Permissions. Chalice associates permissions with memory locations, as described in the introduction, and the permissions held by a method activation can be fractional [4]. The Chalice language builds in specification constructs, such as the pre- and postconditions shown in the example above. The specifications are written in the style of implicit dynamic frames [20], which means they include *accessibility predicates*. Accessibility predicates instigate the transfer of permissions at the point where the specification takes effect. For example, a method precondition **acc(this.f, π)** says that, at the time of a call to the method, a (strictly positive) fraction π of the permission to **this.f** is transferred from caller to callee, and the caller satisfies the precondition only if it possesses at least this amount of permission to **this.f** at the time of the call.

To keep track of permissions, the formalization of Chalice [12] associates with each method activation a *permission mask*, that is, a map `Mask` from locations to the method activation's permission to that location. We assume here that permission amounts are represented by rational numbers, but other structures are also possible. Initially, the permission mask is empty: that is, 0 for all locations.

The semantics of read and write statements in Chalice include proof obligations that the current method's mask contains the necessary permissions. So for a read access to `e.f`, the verifier checks `Mask[`$[\![$`e`$]\!]$`,f] > 0`, and for a write access it checks `Mask[`$[\![$`e`$]\!]$`,f] = 1`, where $[\![$`e`$]\!]$ represents the translation of a Chalice expression into an appropriate Boogie expression.

Permission Transfer. The treatment of specifications is formalized using two operations: *inhale* and *exhale*. Analogously to sequential verification, in which a method precondition is checked at a call site and assumed inside the method body, Chalice says that the caller exhales the precondition and the callee inhales it, and vice versa for the postcondition.

The inhale and exhale operations are defined recursively over the syntax of specifications. Each sub-expression is encoded as a sequence of Boogie statements, which are composed sequentially. Conditions in a specification (such as implications) are encoded as conditional statements.

For an expression `e` not containing accessibility predicates, inhaling `e` means **assume** ⟦`e`⟧, where **assume** indicates a condition that the verifier is allowed to assume, and exhaling `e` means **assert** ⟦`e`⟧, where **assert** indicates a proof obligation. Inhaling an accessibility predicate **acc**`(e.f, p)` adds the permission amount `p` to the permission mask:

```
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] + ⟦p⟧;
```

Exhaling **acc**`(e.f, p)` checks that the mask contains at least the permission amount `p` and then removes `p` from the permission mask:

```
assert Mask[⟦e⟧, f] ≥ ⟦p⟧;
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - ⟦p⟧;
```

In addition, exhaling an expression assigns an arbitrary value to those *heap* locations for which the mask contains no permission after the exhale. This "havoc" operation models possible state updates by other method activations, including those in different threads. It says that if a memory location `e.f` is no longer readable by the current method activation, then any knowledge about its value is removed. In contrast, if the current method activation still holds a non-zero permission to `e.f` after the exhale, then no other method activation can have the full (write) permission to `e.f`, and so whatever the current method activation already knows about the value `e.f` can be soundly retained; that is, can be framed.

Because of the recursive definition of exhale, exhaling an expression such as **acc**`(e.f, p)` `&&` **acc**`(e.f, p)` will exhale **acc**`(e.f, p)` *twice*; the expression is essentially equivalent to **acc**`(e.f, 2p)` (the conjunction behaves *multiplicatively* with respect to permissions [18]). This conjunction can be formally related to the separating conjunction of separation logic [18].

4 Abstract Read Permissions

In this paper, we propose *abstract read permissions*, which allow the programmer to reason abstractly at the level of read and write permissions rather than concrete fractions. In this section, we introduce the main ideas behind abstract read permissions in method specifications. We discuss details of the encoding in Sec. 6 and extensions to other forms of specifications in Sec. 8.

We use two main kinds of accessibility predicates. A full permission (corresponding to a "1"-valued fractional permission) to a location `e.f` is denoted by **`acc(e.f, 1)`** and allows a method to both read and write the location. To specify read permissions, we introduce an accessibility predicate of the form **`acc(e.f, rd)`**, called an **`rd`**-*predicate*. The *abstract permission expression* **`rd`** denotes a positive amount of permission to `e.f` and, thus, permits read access, regardless of what the actual amount is. Every occurrence of **`rd`** in the specification of *one* method invocation denotes the same permission amount (regardless of the location mentioned), but the particular amount may be different for other method invocations. In particular, different activations of a recursive method may interpret **`rd`** differently.

Compared with the options mentioned in Sec. 2, our abstract permission expression **`rd`** is most similar to a ghost parameter that is passed to the method. But instead of the programmer having to compute the amount to be passed in, our approach handles the specification of the amount automatically. Intuitively, the positive amount chosen is small enough for the caller to handle (that is, it stays within a budget of the permissions that the caller has) and small enough that it does not completely rob the caller of all permissions to a memory location (thus enabling framing). The program verifier will produce an error if such a value does not exist.

Note that the exact amount chosen for a call is never revealed in our permission model. Indeed, as we shall see, we encode the amount as a symbolic value that satisfies certain constraints. The main challenge in our design is to identify where and how we should constrain the amount for an **`rd`**-predicate.

Example. Methods frequently require some permission to a location `e.f` and return this permission to their callers. That is, both the pre- and postcondition mention some **`rd`**-predicate such as in method `foo` in Listing 5. Because both occurrences of **`rd`** in the specification of `foo` denote the same amount of permission, method `main` is able to recombine this permission to obtain full permission again (as required by the assignment that follows the call). This part of the example motivates our design decision that every occurrence of **`rd`** in one method specification is interpreted as the same permission amount. Furthermore, the permission amount automatically chosen for the call to `foo` is strictly less than what the caller has. Therefore, the caller retains some permission to `c.val` across the call, which enables framing. In our example, this implies that the value of `c.val` cannot be changed by the call to `foo`; thus, the right-hand side of the last assignment to `c.val` evaluates to 5 (as required by the postcondition of `main`).

4.1 Method Implementations

For the verification of each *method implementation*, we introduce a new *permission constant* π_{method}, which is used to interpret every **`rd`**-predicate in the specification of that method for every location. No precise value is given to π_{method}; we assume only that it is a proper read permission: $0 < \pi_{method} < 1$. The method is then verified as usual in Chalice: we inhale the precondition, execute the method

```
method main(c: Cell)
  requires acc(c.val, 1)
  ensures acc(c.val, 1) && c.val = 5
{
  c.val := 0
  call foo(c)
  c.val := c.val + 5
}
method foo(c: Cell)
  requires acc(c.val, rd)
  ensures acc(c.val, rd)
{ /* ... */ }
```

Listing 5. A simple example that illustrates the choice to interpret all **rd**-predicates as the same fraction in a method specification

body, and then exhale the postcondition. Because the assumption about π_{method} is so weak, a successful verification of the method implementation accommodates any permission amount between 0 and 1 chosen for the call and transferred by the caller. Of course, as usual, only those pre-states and parameter values — and now also the value of **rd** — that can feasibly satisfy the precondition need to be considered when verifying the implementation. For example, for a precondition `acc(e.f, rd) && acc(e.f, rd)`, the precondition can be satisfied only for values of **rd** that are bounded by `.5`.

4.2 Method Calls

A call to a method `m` is verified using `m`'s specification, which may mention **rd**-predicates. Since we verify that the implementation of `m` is correct for any permission amount that one might use to interpret these predicates, the caller is free to choose any fraction between 0 and 1 to interpret them. But we must take care when constraining this choice automatically, because if the constraints are too weak then it will cause callers to fail to verify, and our system would not be practical to use.

If the called method `m` requires an **rd**-predicate to some location `e.f`, we need to check only that the caller holds a positive amount of permission to `e.f`. If it does, intuitively we can always find a (positive) fraction that is smaller than the held amount, and we can transfer this fraction to the callee. This idea is reflected in the encoding of method calls as follows.

We first introduce a permission constant π_{call}, which is used to interpret every **rd**-predicate in the specification of the callee `m` for every location. π_{call} is constrained to be strictly positive (via a Boogie **assume** statement). When exhaling `m`'s precondition, we further constrain π_{call} to be smaller than the permission amount currently held by the caller, for any location for which `m` requires an **rd**-predicate. More precisely, for each **rd**-predicate to be exhaled (that is, each occurrence of `acc(e.f, rd)` for some `e.f`), we first check that the caller

has a positive amount of permission to `e.f`, and we constrain π_{call} to be strictly smaller than this positive amount. Next, we subtract π_{call} from the permission mask (which we can do symbolically even if there will be further constraints on the value of π_{call}) and continue exhaling the precondition. This encoding is reflected in the following Boogie code:

```
assert Mask[⟦e⟧, f] > 0;
assume πcall < Mask[⟦e⟧, f];
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - πcall;
```

The encoding ensures that the callee `m` is provided with the required read permission for each relevant location, while `m`'s caller also retains read permissions to those locations. The latter lets the caller prove that `m` does not modify those locations; that is, it enables framing.

Note that our design interprets all **rd**-predicates in a method specification as the *same* permission amount, regardless of the corresponding memory location. This is not a restriction, because the amount is always implicitly chosen to be smaller than any corresponding amount held by the caller. Also, note that if the specification mentions multiple **rd**-predicates to the same location, then we effectively choose an amount that is small enough such that giving away all of those **rd**-predicates is allowed. This is achieved by constraining π_{call} multiple times, once for every **rd**-predicate. The main soundness argument shows that the generated constraints are satisfiable: see Sec. 7.

After exhaling the callee's precondition, our encoding of a call inhales the postcondition, using the same value π_{call} to interpret any **rd**-predicates mentioned. This allows the caller to regain the same amount of permission that it gave away, if **rd** is mentioned in both the pre- and postcondition. So, in the example from Listing 5, method `main` regains write permission to `c.val` after the call to `foo` and, thus, may write to the field.

Example. To illustrate abstract read permissions, we revisit method `eval` in Listing 1, with the placeholder π replaced by the abstract permission expression **rd**. When the method body of `eval` in Listing 2 is verified, the permission to `s.map` starts out as π_{method} after inhaling the precondition. The recursive call to `left.eval` succeeds by exhaling a strictly smaller fraction, which is then regained on inhaling `eval`'s postcondition. Analogously, a second fraction is given away and regained for the second recursive call. Thus, the permission to `s.map` at the end of the method is again π_{method}, which is required to successfully exhale the postcondition.

This example illustrates that abstract read permissions do not require the overhead of the ghost-parameter option in Sec. 2; a programmer neither has to declare ghost parameters nor provide concrete values for ghost parameters when a method is called. Moreover, they address the first three of the four shortcomings of concrete fractions and counting permissions that we discussed in Sec. 2: (1) Since the permission amount chosen is context-dependent, it is adjusted for each call, which lends itself to flexible re-use. (2) The specification expresses only which read and

write permissions are requested, but not any concrete permission amounts. Therefore, they do not contain any irrelevant information. (3) For a call, **rd**-predicates are constrained to be strictly smaller than the permission held by the caller, which allows framing. In particular, `s.map` $\neq$ **null** is still known to hold after the first recursive call because the caller retains some permission to `s.map` during the call, which allows the verifier to prove the precondition of the second call. Abstract read permissions also address shortcoming (4), as we discuss next.

4.3 Asynchronous Method Calls

Chalice supports asynchronous method calls, using **fork** and **join** statements. A statement `tk` := **fork** `m()` forks off a new thread that executes the method `m`, and returns a *token* which can be used to join the thread and wait on the result `r` of the call, using a statement `r` := **join** `tk`. The verification of asynchronous calls is analogous to synchronous calls, but with the inhale and exhale separated: when a thread is forked to execute a method, the method's precondition is exhaled; at the time a forked thread is joined, the postcondition of the corresponding method (which in Chalice is determined by the type of the token) is inhaled.

For the same reasons as for synchronous method calls, it is useful for asynchronous calls to interpret all **rd**-predicates in a method specification as the same permission amount. In particular, if a **fork** and its corresponding **join** occur in a scoped fashion (in the same method body), we would like to be able to express that we can match up the same permission amounts from corresponding **rd**-predicates. We encode this by adding a ghost field to tokens, which represents the permission amount used to interpret **rd**-predicates for the associated asynchronous call. We record this value in the ghost field when a token is created at a **fork** statement, and refer to it when interpreting the permissions returned at a **join**. This value is never changed; if we encounter the **fork**/**join** statements for a token on the same path through a method body, the same permission fraction is known to be used for both. However, if the **join** takes place in a different method body, no information will be known about this fraction; it is effectively arbitrary (although positive). In Sec. 5, we show how to avoid this loss of information through additional specifications.

Example. Let us again consider method `eval` from Listing 1, with the placeholder π replaced by the abstract permission expression **rd**, and the parallel implementation from Listing 3. The verification of the parallel implementation is performed as follows. First, the precondition of `Add.eval` is inhaled, adding π_{method} to the mask for the location `s.map`. Then, at the first **fork** statement, our encoding introduces a fresh constant $\pi_{fork1} > 0$ to interpret all **rd**-predicates in the specification of `eval` for the first **fork** statement. When exhaling the precondition of the forked method, permission for `s.map` is (successfully) checked to be positive, and π_{fork1} is constrained to be strictly smaller than the currently held amount. Consequently, when π_{fork1} is transferred to the new thread, the forking thread still holds a positive permission amount for `s.map`. The verification of the second **fork** statement is analogous, with another constant $\pi_{fork2} > 0$. So after

the two **fork** statements, the forking method is left with $\pi_{method} - \pi_{fork1} - \pi_{fork2}$. When inhaling the postcondition at the two **join** statements, the fractions π_{fork1} and π_{fork1} are regained; the verifier knows that these amounts are the same as for the two **fork** statements, since the amounts were recorded in a ghost field of the two tokens. Consequently, the forking method `Add.eval` now holds π_{method}, which allows it to exhale its postcondition.

This example illustrates that abstract read permissions enable flexible splitting of permissions. Since permission amounts are constrained to be strictly smaller than the amounts held by the current method activation, abstract read permissions even support *unbounded* permission splitting. That is, they also address the final shortcoming (4) of concrete fractions and counting permissions discussed in Sec. 2. Note that, in contrast to the ghost-parameter option from Sec. 2, programmers do not have to devise a strategy to determine how to split permissions (for instance by splitting the currently held permission in half whenever a fraction needs to be transferred) and how to re-combine the permissions, which is tricky when the order in which the permissions are regained is not statically known.

4.4 Losing Permission

Using the same permission amount to interpret all **rd**-predicates in a method specification is useful for most implementations. However, this can be too restrictive when a method `m` gives away some permission to a location (for instance, during a **fork**) and returns what is left. To handle these situations, we introduce an alternative abstract read expression **rd**$*$, which gets interpreted as another positive, but otherwise unrelated permission amount. In particular, there is no guarantee that it corresponds to the same amount as any other permission expression used in the program. So we could specify `m` to require an **rd**-predicate and to ensure an **rd**$*$-predicate.

When inhaling an **rd**$*$-predicate, no information is assumed about the permission amount it denotes, other than it being positive. Therefore, we can exhale an **rd**$*$-predicate by checking that the current method activation has *some* permission to the appropriate location, and then interpreting the **rd**$*$-predicate as a strictly smaller amount.

5 Permission Expressions

Our design so far provides only two ways of specifying read permissions, **rd**-predicates and **rd**$*$-predicates. The resulting expressiveness is insufficient for some interesting examples. Consider for instance a method `m` that requires full permission to some location, transfers an **rd**-predicate to a newly forked thread `tk`, and returns the remaining permission (along with the token `tk`) to its caller. So far, we can specify that `m` returns *some* permission to the caller using an **rd**$*$-predicate, but we have no way of denoting the precise permission amount it returns (the *difference* between two permission amounts). However, the precise information is necessary for the caller to regain full permission by joining `tk`.

To provide sufficient expressiveness for such examples, we generalise the accessibility predicates **acc(e.f, 1)** and **acc(e.f, rd)** to the new form **acc(e.f, p)**, where p is a *permission expression*. Permission expressions p are defined inductively as follows:

$$
\begin{array}{lll}
p ::= & \texttt{c} & \textit{concrete fraction} \\
\mid & \textbf{rd} & \textit{abstract read permission} \\
\mid & \textbf{rd(tk)} & \textit{token read permission} \\
\mid & p_1 + p_2 & \textit{permission addition} \\
\mid & p_1 - p_2 & \textit{permission subtraction} \\
\mid & \texttt{n} * p & \textit{permission multiplication}
\end{array}
$$

where **c** is a rational literal ($0 < \texttt{c} \leq 1$), **tk** is a token, and **n** is an integer-valued expression

The literal permission expression **c** subsumes full permissions, and **rd** is used as before. The expression **rd(tk)** refers to the amount of permission associated with an **rd**-expression for a particular asynchronous call, via its token. Finally, we support addition, subtraction, and integer multiplication of permission expressions. This allows us in particular to specify the exact permission amount returned by method **m** above, using the permission expression **1 - rd(tk)** in its postcondition.

In addition to the generalised accessibility predicates, we continue to support **rd***-predicates. However, we decided not to support permission expressions containing **rd*** because the meaning of such expressions is sometimes un-intuitive (for instance, **1 - rd* + rd*** does not necessarily denote a full permission since the two occurrences of **rd*** may be interpreted differently) and because they do not provide extra expressiveness (for instance, **1 - rd*** and **rd*** express the same information, namely an unknown, positive permission amount).

6 Encoding

The introduction of permission expressions leads to new subtleties in the encoding of abstract read permissions. This section revises the encoding sketched in Sec. 4 to address these subtleties.

First, because permission expressions p can include subtraction and multiplication, it is possible to write expressions such as **rd-1** that are not guaranteed to denote valid (that is, positive) permission amounts. Exhaling such permission amounts naïvely could result in a total permission of more than 1 in a method activation, leading to unsoundness. Therefore, we impose an additional well-formedness constraint that each permission expression p must provably denote a strictly positive amount of permission. The exact implementation of this check varies for different kinds of permission expression, as shown later in this section.

Second, permission expressions require more sophisticated rules for constraining the permission constants π_{call} during exhale operations. So far, exhaling an **rd**-predicate led to an upper bound on π_{call} by assuming that π_{call} is *smaller* than the positive amount currently stored in the mask for a particular location.

```
method test(c: Cell)
  requires acc(c.f)
  ensures acc(c.f, rd*) && c.f = 3
{
  c.f := 3;
  call bar(c);
}

method bar(c: Cell)
  requires acc(c.f, 1-rd) && acc(c.f, rd)
{
  c.f := 4;
}
```

Listing 6. Occurrences of **rd** in negative positions need to be treated with care to avoid inconsistencies

With the introduction of permission expressions, the abstract permission expression **rd** can also occur in *negative* positions, for instance in **acc(e.f, 1-rd)**. If we were to adopt the same behaviour when exhaling accessibility predicates with **rd** in negative positions, then we also impose lower bounds, leading to potentially unsatisfiable assumptions.

Exhaling negative occurrences of **rd** may also lead to difficulties with subsequent *positive* occurrences, as the example in Listing 6 shows. At the call to `bar` in the body of `test`, the mask contains full permission to `c.f`. Exhaling the first conjunct of `bar`'s precondition leaves us with exactly π_{call}. When the second conjunct **acc(c.f, rd)** is exhaled, the encoding from Sec. 4 checks that some permission is available and then assumes π_{call} to be strictly smaller than that amount. This would lead to the inconsistent assumption $\pi_{call} < \pi_{call}$.

In our solution to these difficulties, we differentiate between three different types of permission expressions:

Type 1: Those in which **rd** does not occur (e.g., **1-rd(tk)**).
Type 2: Those in which **rd** occurs, but only in positive positions (e.g., **rd-rd(tk)**).
Type 3: Those in which **rd** occurs in negative position(s) (e.g., **1-rd**).

We classify accessibility predicates **acc(e.f, p)** in the same way, according to the type of `p`. The special **rd***-predicates are handled like type-2 accessibility predicates, but with respect to a fresh permission constant for each occurrence.

We now describe how to encode the exhale of a precondition at a (synchronous or asynchronous) method call, in terms of how we generate appropriate Boogie code. Inhales are encoded as described earlier; no constraints on π_{call} are generated. To encode a method call, we first introduce a fresh permission constant π_{call} and constrain it to denote a proper read permission:

havoc π_{call}; **assume** $0 < \pi_{call} < 1$;

If the method call is asynchronous, we additionally store π_{call} in a ghost field of the corresponding token. (Since the value of this ghost field never changes, no permission management is necessary for that field.) Then, we exhale the precondition in three phases, each handling one type of accessibility predicate.

Phase 1: The first phase handles logical expressions without accessibility predicates and those accessibility predicates that denote permission amounts that were already fixed before the call; that is, predicates of type 1. To do this, we pass over the precondition, generating assertions for all logical expressions, and ignoring all accessibility predicates of types 2 and 3. For each accessibility predicate **acc**`(e.f, p)` of type 1, we encode the exhale as described in Sec. 3; that is, by generating the following code:

assert $[\![$`p`$]\!] > 0$;
assert `Mask[`$[\![$`e`$]\!]$`, f]` $\geq [\![$`p`$]\!]$;
`Mask[`$[\![$`e`$]\!]$`, f]` := `Mask[`$[\![$`e`$]\!]$`, f] -` $[\![$`p`$]\!]$;

Phase 2: The second phase generates constraints on π_{call} that ensure that accessibility predicates of type 2 can be exhaled (if possible), leaving some remainder. To do this, we pass over the precondition again, ignoring logical expressions and accessibility predicates of types 1 and 3. For each accessibility predicate **acc**`(e.f, p)` of type 2, we assume a rewriting of the form `p = p' + n *` **rd** (for `n` > 0) such that `p'` is some permission expression not mentioning **rd**. We then generate code that constrains the value of π_{call}:

assert $[\![$`p'`$]\!] \geq 0$;
assert `Mask[`$[\![$`e`$]\!]$`, f] >` $[\![$`p'`$]\!]$;
assume `n *` π_{call} `< (Mask[`$[\![$`e`$]\!]$`, f] -` $[\![$`p'`$]\!]$`);`
`Mask[`$[\![$`e`$]\!]$`, f]` := `Mask[`$[\![$`e`$]\!]$`, f] - (`$[\![$`p'`$]\!]$ `+ n *` π_{call}`);`

Phase 3: The third phase exhales the remaining accessibility predicates, without introducing further constraints on π_{call}. To do this, we pass over the precondition a third time, ignoring logical expressions and accessibility predicates of types 1 and 2. For each accessibility predicate of type 3, we generate the same code as in Phase 1.

Constraining π_{call} after accessibility predicates of type 1 have been exhaled results in stronger assumptions, since we assume the value of π_{call} is smaller than the amounts held after Phase 1 is finished. A permission expression of type 2 comes with the requirement that the part that does not mention **rd** (the `p'` above) is non-negative; thus, the whole expression denotes a positive amount. Exhaling accessibility predicates of type 3 only after all constraints have been generated in Phase 2 solves the problems mentioned at the beginning of this section: exhaling **rd** in negative positions does not generate any constraints and, thus, introduces no lower bounds on π_{call}. Moreover, the precondition of method `bar` (Listing 6) is now exhaled soundly; we first exhale the second conjunct (in Phase 2), which constrains π_{call} and leaves $1 - \pi_{call}$ in the mask, and then exhale the first conjunct (in Phase 3). This does not lead to additional constraints and, thus, does not introduce inconsistent assumptions.

Any conditionals in the precondition are handled in each phase. However, the evaluation of such conditionals are unaffected by our manipulation of permissions, because conditionals in assertions are syntactically restricted not to depend on permissions (assertions such as `acc(x.f)` ⇒ `acc(y.f)` are forbidden).

After all phases are complete, our encoding removes all knowledge about locations to which no permission remains in the mask, as we explained in Sec. 3.

7 Soundness

In this section, we give a brief argument for the soundness of our encoding. A soundness proof for an entire verification methodology using our permission model is beyond the scope of our paper, and for the most part involves arguments that are orthogonal to the contributions of this paper.

Compared to the standard fractional permission model, we introduced abstract read permissions and encode them as underspecified constants in Boogie. The most relevant concern for soundness with respect to this paper is that the assumptions that we introduce about the constants used to interpret abstract permission expressions must not lead to contradictions. Apart from the points in our encoding where these assumptions are generated, abstract read permissions are treated just as any other permission amounts in permission expressions.

A new constant π_{call} to denote the underspecified amount is introduced at each method call in the encoding of the source program. From the end of Phase 2 (as described in the previous section), these amounts are treated just like any other permission amount with a fixed interpretation. Therefore, it is sufficient for us to justify that, for each method call, the assumptions generated in Phase 2 are always satisfiable, provided that none of the assertions in the generated code fail. Operationally (though this is never required in the verifier), one can think of this amount as being chosen (by some oracle) at the point of the method call, in such a way that the assumptions are all satisfied; we just need to justify that this is possible. As we show below, each assumption during Phase 2 imposes a (strictly positive) upper bound on the possible values of π_{call}. Since the only lower bound imposed is 0 and since we assume that permission amounts are rational numbers, the assumptions are then guaranteed to be satisfiable. In the following, we focus on the permission expressions presented in Sec. 5, but the arguments apply equally to `rd`$*$-predicates.

Let us consider the exhale of the method precondition for any given method call, and let π_{call} be the permission constant introduced for that call. Let us further consider the amount of permission (for any location) stored in the `Mask` up to the start of Phase 2 of exhaling the precondition as a formula representing the arithmetic performed so far during the verification. We first apply an inductive argument that during Phase 2 of the exhale, this formula remains expressible in the form $\rho - \mathtt{m} * \pi_{call}$, for some formula ρ not mentioning π_{call} and some integer $\mathtt{m} \geq 0$: since π_{call} is chosen to be a fresh constant for each call, it is clear that up to the start of Phase 2, the formula representing the current permission amount in the mask for any location cannot depend on the fresh π_{call} constant.

Each exhale of an accessibility predicate during Phase 2 subtracts multiples of π_{call} from the mask, and so the amount stored remains expressible in the form $\rho - \mathtt{m} * \pi_{call}$.

Now consider the handling of an arbitrary type-2 accessibility predicate **acc**(e.f, p) during Phase 2. Just as in the encoding presented in the previous section, we assume a rewriting of the form p = p' + n * **rd** (for some $\mathtt{n} > 0$) such that p' is some permission expression not mentioning **rd**. Using the argument so far, we may assume that there exist an integer $\mathtt{m} \geq 0$ and a formula ρ not mentioning π_{call}, such that:

$$\mathtt{Mask}[[\![\mathtt{e}]\!], \mathtt{f}] = \rho - \mathtt{m} * \pi_{call}$$

We can now use this equality to rewrite the relevant code generated in Phase 2:

assert Mask[⟦e⟧, f] > ⟦p'⟧;
assume n * π_{call} < (Mask[⟦e⟧, f] - ⟦p'⟧);
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - (⟦p'⟧ + n * π_{call});

into the following (equivalent) form:

assert ρ - m * π_{call} > ⟦p'⟧;
assume π_{call} < (ρ - ⟦p'⟧) / (m + n);
Mask[⟦e⟧, f] := ρ - ⟦p'⟧ - (m + n) * π_{call};

Since m and π_{call} are non-negative, the assertion implies that $(\rho - [\![\mathtt{p'}]\!]) > 0$. Combined with the facts $\mathtt{m} \geq 0$ and $\mathtt{n} > 0$, this gives us that $(\rho - [\![\mathtt{p'}]\!])/(\mathtt{m}+\mathtt{n})$ is strictly positive. Therefore, the assumption only imposes an extra strictly positive upper bound on the permitted values of π_{call}. Since this argument applies to each accessibility predicate generated in Phase 2, all assumptions generated impose strictly positive upper bounds on π_{call}, and thus, combined with the only other assumptions about this value, that $0 < \pi_{call} < 1$, the assumptions about π_{call} are always satisfiable.

8 Monitors

Chalice supports monitors, which have an associated *monitor invariant*, describing the permissions held and properties guaranteed while the monitor is unlocked. When a monitor is acquired, the monitor invariant is inhaled, and when the monitor is released, it is exhaled [12]. It can be useful for a monitor invariant to provide read permissions to the fields it describes. A typical example is a single-writer, multiple-reader scenario, which can be handled by splitting a full permission between the writer thread and the monitor. By acquiring the monitor, a thread can obtain read permission. The writer can combine the fraction from the monitor with the fraction it already holds to obtain full permission.

Supporting abstract read permissions for monitor invariants is more difficult than for methods. If we allowed a thread to choose a permission amount for **rd**-expressions when exhaling the monitor invariant upon release (analogously to choosing the amount when exhaling a method precondition) then one could not

soundly assume that the next acquire in the same thread will inhale the same amount — other threads might have acquired and released the monitor in the meantime and interpreted the **rd**-expressions in the monitor invariant differently. Similar issues arise with folding and unfolding abstract predicates [17] as well as with sending and receiving messages [14].

If the threads interacting with a monitor never need to know that the amount of read permission that they inhale from a monitor invariant is related to some other amount earlier in the program execution, then we can employ **rd**$*$-predicates in the monitor invariant and handle them just as for method calls.

However, in situations like the single-writer, multiple-reader example above, we must associate a persistent amount of permission with a monitor invariant to allow the writer thread to obtain full permission. We have considered various solutions to this problem. One is to fix the permission amount that **rd**-expressions are interpreted with "once and for all", either for all monitors (which has the advantage that such permissions can be transferred between monitors), or when a new monitor is created (which has the advantage that the amount can be chosen with respect to what is held at that point). An alternative is to store the permission amount in a ghost field of the object, similarly to the ghost-parameter option in Sec. 2. This is more flexible, but permissions to this ghost field must then be appropriately handled, and information about the field value needs to be communicated in specifications for this to really give an advantage.

In our implementation, we currently take the simplest approach described above; we generate one (underspecified) permission constant to interpret all **rd**-expressions in monitor invariants, abstract predicates, and message invariants. This has been sufficiently expressive, but we are also evaluating the other options.

9 Related Work

Fractional permissions were proposed by Boyland [4]. He uses them in a type system to check non-interference of the branches of a parallel composition and to show that non-interfering parallel compositions have deterministic results. Zhao [21] uses fractional permissions to prevent data races in concurrent Java programs. Neither of the two systems supports the verification of a program w.r.t. to a programmer-supplied specification.

The use of fractional permissions for program verification was first explored in the context of separation logic. Bornat *et al.* [3], Gotsman *et al.* [7], and Hobor *et al.* [9] all employ separation logic with fractional permissions. They use concrete fractions and logical variables in specifications, which has the drawbacks discussed in Sec. 2.

Our original verification methodology for Chalice [12] supports fractional permissions (expressed as integer percentages) and infinitesimal permissions, which are similar to counting permissions. We built on implicit dynamic frames [20], which allows us to generate first-order verification conditions, which can be handled by automatic SMT solvers. The permission model used in this work suffers from the shortcomings discussed in Sec. 2. To solve these problems, we introduced the idea of underspecified, constrained fractions in a workshop paper [8].

The present paper extends our earlier work with more detailed explanations (especially of the encoding) and with a soundness argument.

Other systems have also aimed to package fractions in more abstract ways. In his original work on fractional permissions, Boyland uses a type system that allows permission polymorphism [4]. A non-deterministic type checker determines the possible ways fraction variables used in method signatures can be instantiated. Only a sketch of an algorithm for coming up with the instantiation is provided; it is not clear how the sketched approach deals with repeated fraction variables or with fraction variables occurring in negative positions.

The separation-logic based verifier VeriFast supports fractional permissions and logical variables [10]. When a logical variable specifies a permission amount, there is some limited support to set it automatically at a call site. However, only the first use of the variable is considered (so fractions cannot be correlated) and that first use will soak up all the permission that the caller has (so framing is not supported).

Bierhoff *et al.* [2,16] recently presented fraction-free permission type systems. Their permissions, which have intuitive names such as "unique" and "local immutable", are held by variables, whereas our permissions are held by method activation records (and monitors, etc.). Permission transfers happen at assignments and parameter passing. If the target of the transfer only needs a fraction, the type system automatically carves up the permission held. Permissions that cross method boundaries can be *borrowed* (meaning they will be transferred back upon return of the method) or *consumed*. Borrowing is related to our rule of interpreting all **rd**-expressions in one method specification as the same permission amount, while consuming is related to our **rd***-predicates.

Bierhoff [2] presents a verification system that makes use of permissions as a subsequent step after permission type checking. The integration of both steps in our system provides more expressiveness, for instance, because permissions can be denoted conditionally, using logical implication.

Recent work of Militão et al. [15] generalizes fractional permissions to user-defined *views*, which can describe permissions to sets of fields, and properties such as reference uniqueness. They also employ fractions whose values are hidden from the user, but do not have an analogue to permission expressions (cf. Sec. 5).

There are other ways of specifying that a method will treat something as read-only. The C verifier VCC employs *claims* [5], which can specify a certain set of objects which cannot be modified while the claim exists. Using reference counting, an object keeps track of how many outstanding claims it has. Claims are themselves (ghost) objects, so there can be claims on claims. This allows programs like our `eval` example in Sec. 2 to be specified and verified, but at the cost of having to write the (ghost) code that sets up and destroys the claims.

10 Conclusions

We have presented a novel methodology for specifying sequential and concurrent programs based on fractional permissions. Our abstract read permissions

allow programmers to specify access permissions at the level of read and write permissions, without the need to reason using a concrete mathematical model or syntax. Our methodology avoids shortcomings of working with concrete fractions and, by picking a judicious interpretation for abstract read permissions in method specifications, imposes less specification and verification overhead than solutions based on ghost parameters or logical variables. In cases where the differences between permission amounts are important, our permission expressions provide a natural and abstract way of conveying the relevant information.

We have explained our methodology in terms of implicit dynamic frames and verification condition generation. However, abstract read permissions are independent of any particular permission logic or verification technique. So far, our methodology has been implemented in two verifiers for Chalice: one based on verification condition generation and one on symbolic execution. Both verifiers make use of the support for real numbers in Z3 [6].

Our permission expressions support the addition and subtraction of a bounded number of **rd**-expressions. As future work, we plan to handle the (statically) unbounded case, for instance, by supporting mathematical sums over unbounded sets or sequences. We could then specify a method that forks an unbounded number of threads (each requiring read permission to some shared data) and stores the tokens in a list. By removing tokens from the list, we could easily support a specification for rejoining the threads, regardless of the order of joins. This kind of example would be difficult to support with concrete fractional permissions.

We are also exploring other possible uses of abstract read permissions; exploiting the use of permission amounts which can be freely constrained (from above). We are considering the possibility of manually introducing such amounts in other verification situations, and also basing an approach to handling immutable data on this novel specification concept.

Acknowledgements. We would like to thank the attendees and reviewers of the Formal Techniques for Java-like Programs 2011 workshop, as well as the attendees of the Dublin Concurrency Workshop 2011, particularly Andrew Butterfield and Peter O'Hearn, for encouraging feedback on a preliminary presentation of this work. We thank John Boyland for discussions of the comparisons with fractional permissions. We are grateful to Malte Schwerhoff for many discussions about the details of our model and to Martin Vechev for useful feedback on a draft of this paper.

References

1. 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)
2. Bierhoff, K.: Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning. In: ONWARD, pp. 19–32. ACM (2011)

3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270. ACM (2005)
4. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
5. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local Verification of Global Invariants in Concurrent Programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010)
6. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local Reasoning for Storable Locks and Threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
8. Heule, S., Leino, K.R.M., Müller, P., Summers, A.J.: Fractional permissions without the fractions. In: Formal Techniques for Java-like Programs, FTfJP (2011)
9. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle Semantics for Concurrent Separation Logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
10. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
11. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing Verification Condition Generation with Symbolic Execution: An Experience Report. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 196–208. Springer, Heidelberg (2012)
12. 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)
13. Leino, K.R.M., Müller, P., Smans, J.: Verification of Concurrent Programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
14. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-Free Channels and Locks. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 407–426. Springer, Heidelberg (2010)
15. Militão, F., Aldrich, J., Caires, L.: Aliasing control with view-based typestate. In: FTfJP, pp. 7:1–7:7. ACM (2010)
16. Naden, K., Bocchino, R., Aldrich, J., Bierhoff, K.: A type system for borrowing permissions. In: POPL, pp. 557–570. ACM (2012)
17. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL. ACM (2005)
18. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: Logical Methods in Computer Science (to appear, 2012)
19. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. IEEE (2002)
20. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
21. Zhao, Y.: Concurrency Analysis based on Fractional Permission System. PhD thesis, The University of Wisconsin–Milwaukee (2007)

Sound and Complete Flow Typing with Unions, Intersections and Negations

David J. Pearce

Victoria University of Wellington
Wellington, New Zealand
djp@ecs.vuw.ac.nz

Abstract. Flow typing is becoming a popular mechanism for typing existing programs written in untyped languages (e.g. JavaScript, Racket, Groovy). Such systems require intersections for the true-branch of a type test, negations for the false-branch, and unions to capture the flow of information at meet points. Type systems involving unions, intersections and negations require a subtype operator which is non-trivial to implement. Frisch *et al.* demonstrated that this problem was decidable. However, their proof was not constructive and does not lend itself naturally to an implementation. In this paper, we present a sound and complete algorithm for subtype testing in the presence of unions, intersections and negations.

1 Introduction

Statically typed programming languages lead to programs which are more efficient and where errors are easier to detect ahead-of-time [1, 2]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. In contrast, dynamically typed languages are more flexible in nature which helps reduce overheads and increase productivity [3–6].

A common complaint against statically typed languages is the need for often unnecessarily verbose type declarations. Hindley-Milner Type inference [7, 8] is a common approach to addressing this problem, where type declarations are inferred automatically. Scala [9], C#3.0 [10] and OCaml [11] provide good examples of this in an imperative setting. However, such languages still require each program variable to have exactly one type. Flow typing offers an alternative to Hindley-Milner type inference where a variable may have different types at different program points. The technique is adopted from flow-sensitive program analysis and has been used for non-null types [12–15], information flow [16–18], purity checking [19] and more [12, 13, 20–26].

Few languages exist which incorporate flow typing directly. Typed Racket [23, 24] provides a *typed* sister language for *untyped* Racket, where flow typing is used to capture common idioms in the untyped language. Similarly, the recent 2.0 release of the popular Groovy language includes a flow typing algorithm [27]. Again, this is designed to handle common idioms in (previously) untyped Groovy programs. Finally, the Whiley language employs flow-typing to give it the look-and-feel of an untyped language [28–30].

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 335–354, 2013.

1.1 Flow Typing

A defining characteristic of flow typing is the ability to *retype* a variable — that is, assign it a completely unrelated type. The JVM Bytecode Verifier [31], perhaps the most widely-used example of a flow typing system, provides a good illustration:

```
public static float convert(int):
  iload 0    // load register 0 on stack
  i2f        // convert int to float
  fstore 0   // store float to register 0
  fload 0    // load register 0 on stack
  freturn    // return value on stack
```

In the above, register 0 contains the parameter value on entry and, initially, has type `int`. The type of register 0 is subsequently changed to `float` by the `fstore` bytecode. To ensure type safety, the JVM bytecode verifier employs a typing algorithm based upon dataflow analysis [32]. This tracks the type of a variable at each program point, allowing it easily to handle the above example.

Flow typing can also retype variables after conditionals. A *non-null type system* (e.g. [12–15]) prevents variables which may hold `null` from being dereferenced. The following illustrates:

```
int cmp(String s1, @NonNull String s2) {
  if(s1 != null) {
    return s1.compareTo(s2);
  } else {
    return -1;
} }
```

The modifier `@NonNull` indicates a variable definitely cannot hold `null` and, hence, that it can be safely dereferenced. To deal with the above example, a non-null type system will retype variable `s1` to `@NonNull` on the true branch — thus allowing it to type check the subsequent dereference of `s1`.

Whiley [28–30] employs a flow type system to give it the look-and-feel of a dynamically typed language. Variable retyping through conditionals is supported using the `is` operator (similar to `instanceof` in Java) as follows:

```
define Circle as {int x, int y, int r}
define Rect as {int x, int y, int w, int h}
define Shape as Circle | Rect

real area(Shape s):
    if s is Circle:
        return PI * s.r * s.r
    else:
        return s.w * s.h
```

A `Shape` is either a `Rect` or a `Circle` (which are both record types). The type test "`s is Circle`" determines whether `s` is a `Circle` or not. Unlike Java, Whiley automatically retypes `s` to have type `Circle` (resp. `Rect`) on the true (resp. false) branches of the `if` statement. There is no need to explicitly cast variable `s` to the appropriate `Shape` before accessing its fields.

1.2 Unions, Intersections and Negations

Union types (e.g. $\mathtt{T}_1 \vee \mathtt{T}_2$) are commonly used in flow typing systems to capture the type of variables at meet points. For example, consider this code snippet:

```
if ...:
    x = 1
else:
    x = true
...
```

After the assignment `x=1`, the type of variable `x` is `int`. Likewise, after the assignment `x=true` it is `bool`. Finally, `x` has type `int` $\vee$ `bool` immediately after the **`if`** statement (i.e. at the meet point). This indicates `x` can hold either an `int` *or* a `bool` at that point.

Retyping variables after runtime type tests is typically achieved through a type system which supports both *intersections* (e.g. $\mathtt{T}_1 \wedge \mathtt{T}_2$) and *negations* (e.g. $\neg\mathtt{T}_1$). For example, consider again this code snippet:

```
real area(Shape s):
    if s is Circle:
        return PI * s.r * s.r
    else:
        return s.w * s.h
```

To determine the type of variable `s` on the true branch, we *intersect* its declared type (i.e. `Shape`) with the type test (i.e. `Circle`). Likewise, on the false branch, we compute the difference of these two types (i.e. `Shape - Circle`). Observe that the difference of two types in such a system is given by: $\mathtt{T}_1 - \mathtt{T}_2 \equiv \mathtt{T}_1 \wedge \neg\mathtt{T}_2$.

1.3 Contributions

Subtype testing (i.e. establishing whether $\mathtt{T}_1 \leq \mathtt{T}_2$ holds or not) is a challenging algorithmic problem for a type system involving unions, intersections and negations. In particular, we desire that subtyping is both *sound* and *complete* with respect to a semantic model where types are viewed as sets. The former requires $\mathtt{T}_1 \leq \mathtt{T}_2$ holds *only* when $\mathtt{T}_1$ is a subset of $\mathtt{T}_2$, whilst the latter requires that $\mathtt{T}_1 \leq \mathtt{T}_2$ holds *whenever* $\mathtt{T}_1$ is a subset of $\mathtt{T}_2$. Frisch *et al.* demonstrated that this problem was decidable [33]. However, their proof was not constructive and does not lend itself naturally to an implementation. In this paper, we present a sound and complete algorithm for subtyping in the presence of unions, intersections and negations. This contrasts with previous flow type systems (e.g. [23, 24]) which are shown sound, but not complete.

2 A Flow-Typing Calculus — FT

We now introduce our flow-typing calculus, FT, within which we frame our flow typing problem. The calculus is specifically kept to a minimum to allow us to succinctly capture the important issues. In this section, we introduce the syntax, semantics and subtyping rules for FT. We tacitly assume at this point that an appropriate subtyping operator exists. Subsequently, in §3 and §4, we will detail the algorithms which implement this operator (and which are the core contribution of this paper).

2.1 Types

The following gives a *syntactic* definition of types in FT:

$$\mathtt{T} ::= \mathtt{any} \mid \mathtt{int} \mid (\mathtt{T}_1, \ldots, \mathtt{T}_n) \mid \neg\mathtt{T} \mid \mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n \mid \mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n$$

Here, any represents $\top$, int the set of all integers and $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ represents tuples with one or more elements. The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. Union types are generally useful in flow typing systems, as they can characterise types generated at meet points in the control-flow graph. The intersection $\mathtt{T}_1 \wedge \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *and* $\mathtt{T}_2$. Intersections are needed in our flow type system to capture the type of a variable (e.g. x) after a type test (e.g. x is T). The type $\neg\mathtt{T}$ is the *negation* type containing those values *not* in T. Thus, $\neg$any represents $\bot$ (i.e. the empty set) and we will often write void as a short-hand for this. Negations are also useful for capturing the type of a variable on the false branch of a type test. Finally, we make some simplifying assumptions regarding unions and intersections: *namely, that elements are unordered and duplicates are removed*. Thus, $\mathtt{T}_1 \vee \mathtt{T}_2$ is indistinguishable from $\mathtt{T}_2 \vee \mathtt{T}_1$. Likewise, $\mathtt{T}_1 \vee \mathtt{T}_1$ is not distinguishable from $\mathtt{T}_1$. Whilst these simplifications are not strictly necessary, they simplify our presentation. Furthermore, they can be implemented easily enough by sorting elements according to a fixed total ordering of types.

To better understand the meaning of types in FT, it is helpful to give a *semantic interpretation* (following e.g. [33–36]). The aim is to give a set-theoretic model where subtype corresponds to subset. The *domain* $\mathbb{D}$ of values in our model consists of the integers and all records constructible from values in $\mathbb{D}$:

$$\mathbb{D} = \mathbb{Z} \cup \left\{ (\mathtt{v}_1, \ldots, \mathtt{v}_n) \mid \mathtt{v}_1 \in \mathbb{D}, \ldots, \mathtt{v}_n \in \mathbb{D} \right\}$$

Definition 1 (Type Semantics). *Every type* T *is characterised by the set of values it accepts, given by* $[\![\mathtt{T}]\!]$ *and defined as follows:*

$$\begin{aligned}
[\![\mathtt{any}]\!] &= \mathbb{D} \\
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!] &= \{(\mathtt{v}_1, \ldots, \mathtt{v}_n) \mid \mathtt{v}_1 \in [\![\mathtt{T}_1]\!], \ldots, \mathtt{v}_n \in [\![\mathtt{T}_n]\!]\} \\
[\![\neg\mathtt{T}]\!] &= \mathbb{D} - [\![\mathtt{T}]\!] \\
[\![\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cap \ldots \cap [\![\mathtt{T}_n]\!] \\
[\![\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cup \ldots \cup [\![\mathtt{T}_n]\!]
\end{aligned}$$

It is important to distinguish the *syntactic* representation from the *semantic* model of types. The former corresponds (roughly speaking) to a physical machine representation, whilst the latter is a mathematical ideal. As such, the syntactic representation diverges from the semantic model and, to compensate, we must establish a correlation between them. For example int and $\neg\neg$int have distinct syntactic representations, but are semantically indistinguishable. Similarly for $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ and $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$.

Ultimately, we want to construct a subtyping algorithm that is both *sound* and *complete* (i.e. that $\mathtt{T}_1 \leq \mathtt{T}_2 \iff [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$). The distinction between syntactic and semantic forms presents a significant challenge in doing this.

Syntax:

$\mathtt{t} ::=$	*terms:*
$\mathtt{x}$	*variable*
$(\mathtt{t}_1, \ldots, \mathtt{t}_n)$	*tuple*
$\mathtt{f}\ \mathtt{t}_1$	*application*
$\mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$	*declaration*
$\mathtt{if}(\mathtt{x}\ \mathtt{is}\ \mathtt{T})\ \mathtt{t}_1\ \mathtt{else}\ \mathtt{t}_2$	*type test*
$\mathtt{v} ::=$	*values:*
$\mathtt{i}$	*integer*
$(\mathtt{v}_1, \ldots, \mathtt{v}_n)$	*tuple*

Operational Semantics:

$$\frac{\Delta \vdash \mathtt{t}_k \longrightarrow \mathtt{t}'_k}{\Delta \vdash (\ldots, \mathtt{t}_k, \ldots) \longrightarrow (\ldots, \mathtt{t}'_k, \ldots)} \quad \text{(E-TUP)}$$

$$\frac{\Delta \vdash \mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\Delta \vdash \mathtt{f}\ \mathtt{t}_1 \longrightarrow \mathtt{f}\ \mathtt{t}'_1} \quad \text{(E-APP1)}$$

$$\frac{\Delta(\mathtt{f}) = \langle \mathtt{T}, \mathtt{x}, \mathtt{t}_2 \rangle \quad \mathtt{v}_1 \in [\![\mathtt{T}]\!]}{\Delta \vdash \mathtt{f}\ \mathtt{v}_1 \longrightarrow \mathtt{t}_2[\mathtt{x} \mapsto \mathtt{v}_1]} \quad \text{(E-APP2)}$$

$$\frac{\Delta[\mathtt{f} \mapsto \langle \mathtt{T}, \mathtt{x}, \mathtt{t}_1 \rangle] \vdash \mathtt{t}_2 \longrightarrow \mathtt{t}'_2}{\Delta \vdash \mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow \mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}'_2} \quad \text{(E-DEC1)}$$

$$\frac{}{\Delta \vdash \mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{v}_2 \longrightarrow \mathtt{v}_2} \quad \text{(E-DEC2)}$$

$$\frac{\mathtt{v}_1 \in [\![\mathtt{T}]\!]}{\Delta \vdash \mathtt{if}(\mathtt{v}_1\ \mathtt{is}\ \mathtt{T})\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_2} \quad \text{(E-IF1)}$$

$$\frac{\mathtt{v}_1 \notin [\![\mathtt{T}]\!]}{\Delta \vdash \mathtt{if}(\mathtt{v}_1\ \mathtt{is}\ \mathtt{T})\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{t}_3} \quad \text{(E-IF2)}$$

Fig. 1. Syntax and (small-step) operational semantics for FT

2.2 Syntax and Semantics

Figure 1 gives the syntax of FT along with a small-step operational semantics, where $\Delta[\mathtt{f} \mapsto \langle \mathtt{T}, \mathtt{x}, \mathtt{t} \rangle]$ returns Δ with $\mathtt{f}$ now mapped to a triple $\langle \mathtt{T}, \mathtt{x}, \mathtt{t} \rangle$ representing its declaration. Here, $\mathtt{T}$ denotes the parameter type, $\mathtt{x}$ the parameter name and $\mathtt{t}$ the function body. Similarly, $\mathtt{t}[\mathtt{x} \mapsto \mathtt{v}]$ returns the term $\mathtt{t}$ with all occurrences of $\mathtt{x}$ now substituted with $\mathtt{v}$. To avoid issues of variable capture, we assume parameter names are unique and may only occur within their function body (i.e. that for $\mathtt{f}(\mathtt{T}\ \mathtt{x}) = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ parameter $\mathtt{x}$ can only occur in $\mathtt{t}_1$).

From the figure, we see that a semantic notion of type is explicitly required for the operational semantics (as e.g. E-APP2 uses $[\![\mathtt{T}]\!]$). Thus, the semantic notion of execution is separated from the algorithmic notion of subtyping — and our goal in developing a complete subtyping algorithm is to ensure as many correct programs as possible are typeable. The reader may be surprised to see that FT does not include a first-class notion of function value (i.e. a term of the form $\lambda \mathtt{x}.\mathtt{t}$). This avoids a well-known problem of circularity in the definitions (i.e. where the semantic definition of types depends on the operational semantics and vice-versa [33, 36, 37]). In short, including function values adds unnecessary complexity and is therefore omitted. Instead, functions are declared explicitly and a runtime environment, Δ, is used to maintain the mapping from declared functions to their bodies.

An example FT program and its evaluation is given below:

$$\begin{array}{l}
\texttt{f(any x)} = \texttt{if(x is int) 1 else 0} \\
\qquad \texttt{in (f 1, f (1, 2))} \\
\\
\hookrightarrow \texttt{f(any x)} = \texttt{if(x is int) 1 else 0} \\
\qquad \texttt{in (if(1 is int) 1 else 0, f (1, 2))} \\
\\
\hookrightarrow \texttt{f(any x)} = \texttt{if(x is int) 1 else 0} \\
\qquad \texttt{in (1, f (1, 2))} \\
\\
\hookrightarrow \texttt{f(any x)} = \texttt{if(x is int) 1 else 0} \\
\qquad \texttt{in (1, if((1, 2) is int) 1 else 0)} \\
\\
\hookrightarrow \texttt{f(any x)} = \texttt{if(x is int) 1 else 0} \\
\qquad \texttt{in (1, 0)} \\
\\
\hookrightarrow \texttt{(1, 0)}
\end{array}$$

This example illustrates a few interesting aspects of Figure 1. Firstly, for simplicity, the order of evaluation for tuples is undefined under E-TUP. This could easily be specified, but is not important here. Secondly, the term $\texttt{if(x is T) } \texttt{t}_1 \texttt{ else } \texttt{t}_2$ implements a runtime type test (similar to e.g. Java's `instanceof` operator). The left-hand side of this operator is restricted to a variable, rather than a general term. This succinctly captures the problem of retyping a variable within the true (resp. false) branches of the conditional.

2.3 Flow-Typing Rules

The flow-typing rules are given in Figure 2. These are presented as judgements of the form $\Gamma \vdash \texttt{t} : \texttt{T}$, which can be read as saying: *term* $\texttt{t}$ *can be shown to have type* $\texttt{T}$ *under environment* Γ. The environment maps variable names to their current type, and also function names to a pair $\texttt{T}_1 \rightarrow \texttt{T}_2$ capturing the declared parameter and inferred return type. For simplicity, we assume that function names and parameter names do not intersect.

Rules T-INT, T-VAR and T-TUP are straightforward and do not warrant further discussion. The remaining rules are more interesting, and we now consider them in more detail:

- Rule T-APP. For a function application, the type of the argument is determined recursively, whilst the function's declared parameter and inferred return types are obtained from the environment. The rule checks the argument type (i.e. $\texttt{T}_1$) is a subtype of the declared parameter type (i.e. $\texttt{T}_2$) using the subtype operator (i.e. $\texttt{T}_1 \leq \texttt{T}_2$). The subtype operator will be discussed in more detail below.
- Rule T-DEC. For a function declaration, the return type is inferred by typing the body (i.e. $\texttt{t}_2$) using the current environment updated to map the parameter (i.e. $\texttt{x}$) to its declared type (i.e. $\texttt{T}_1$). Using this, the type of the outer term (i.e. $\texttt{t}_3$) is then

Flow-Typing:

$$\frac{\mathtt{v} \in \mathbb{Z}}{\Gamma \vdash \mathtt{v} \ : \ \mathtt{int}} \quad \text{(T-INT)}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} \ : \ \mathtt{T}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{t}_n : \mathtt{T}_n}{\Gamma \vdash (\mathtt{t}_1, \ldots, \mathtt{t}_n) \ : \ (\mathtt{T}_1, \ldots, \mathtt{T}_n)} \quad \text{(T-TUP)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \quad \Gamma(\mathtt{f}) = \mathtt{T}_2 \rightarrow \mathtt{T}_3 \quad \mathtt{T}_1 \leq \mathtt{T}_2}{\Gamma \vdash \mathtt{f}\ \mathtt{t}_1 \ : \ \mathtt{T}_3} \quad \text{(T-APP)}$$

$$\frac{\Gamma[\mathtt{x} \mapsto \mathtt{T}_1] \vdash \mathtt{t}_2 : \mathtt{T}_2 \quad \Gamma[\mathtt{f} \mapsto \mathtt{T}_1 \rightarrow \mathtt{T}_2] \vdash \mathtt{t}_3 : \mathtt{T}_3}{\Gamma \vdash \mathtt{f}(\mathtt{T}_1\ \mathtt{x}) = \mathtt{t}_2\ \mathtt{in}\ \mathtt{t}_3 \ : \ \mathtt{T}_3} \quad \text{(T-DEC)}$$

$$\frac{\Gamma[\mathtt{x} \mapsto \Gamma(\mathtt{x}) \wedge \mathtt{T}_1] \vdash \mathtt{t}_2 : \mathtt{T}_2 \quad \Gamma[\mathtt{x} \mapsto \Gamma(\mathtt{x}) \wedge \neg \mathtt{T}_1] \vdash \mathtt{t}_3 : \mathtt{T}_3}{\Gamma \vdash \mathtt{if}(\mathtt{x}\ \mathtt{is}\ \mathtt{T}_1)\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}_2 \vee \mathtt{T}_3} \quad \text{(T-IF)}$$

Fig. 2. Flow-typing rules for FT

determined. Observe that, under this rule, recursive function calls cannot be typed as $\mathtt{f}$ is not included when typing $\mathtt{t}_2$ — however, this is of little relevance to the problem being addressed.

- Rule T-IF. For a type test, the true and false branches are typed using updated environments. For the true branch, the variable being tested (i.e. $\mathtt{x}$) is mapped to the intersection of its current type and that of the type test (i.e. to $\Gamma(\mathtt{x}) \wedge \mathtt{T}_1$) — this captures the fact that its values are known to be in both $\Gamma(\mathtt{x})$ and $\mathtt{T}_1$. Similarly, for the false branch, the variable being tested is mapped to the intersection of its current type and that of the negated type test (i.e. to $\Gamma(\mathtt{x}) \wedge \neg\mathtt{T}_1$) — this captures the fact that its values are known to be in $\Gamma(\mathtt{x})$ but not in $\mathtt{T}_1$. The resulting type of the type test is then the most precise type which includes the types determined for each branch (i.e. $\mathtt{T}_2 \vee \mathtt{T}_3$).

Observe that, in rule T-IF, the type of the tested variable may be determined as `void` for either branch — which, in such case, indicates that branch is unreachable. A modern compiler would most likely report such a situation as a syntax error (but this is an orthogonal issue).

Discussion. Having considered the flow-typing rules, we can now consider why certain constructs are included in our calculus. Firstly, function application is included since T-App requires a subtype test. Without this construct, there is no need for a subtyping algorithm such as presented in this paper. Secondly, tuple types are included because

Subtyping (incomplete):

$$\frac{}{\mathtt{T} \leq \mathtt{any}}\ [\text{S-ANY1}] \qquad \frac{}{\mathtt{void} \leq \mathtt{T}}\ [\text{S-ANY2}]$$

$$\frac{}{\mathtt{int} \leq \neg(\mathtt{T}_1, \ldots, \mathtt{T}_n)}\ [\text{S-INT1}] \qquad \frac{}{(\mathtt{T}_1, \ldots, \mathtt{T}_n) \leq \neg\mathtt{int}}\ [\text{S-INT2}]$$

$$\frac{\forall \mathtt{i}.\mathtt{T}_i \leq \mathtt{S}_i}{(\mathtt{T}_1, \ldots, \mathtt{T}_n) \leq (\mathtt{S}_1, \ldots, \mathtt{S}_n)}\ [\text{S-TUP1}] \qquad \frac{\mathtt{n} \neq \mathtt{m} \vee \exists \mathtt{i}.\mathtt{T}_i \leq \neg\mathtt{S}_i}{(\mathtt{T}_1, \ldots, \mathtt{T}_n) \leq \neg(\mathtt{S}_1, \ldots, \mathtt{S}_m)}\ [\text{S-TUP2}]$$

$$\frac{\forall \mathtt{i}.\mathtt{T}_i \geq \mathtt{S}_i}{\neg(\mathtt{T}_1, \ldots, \mathtt{T}_n) \leq \neg(\mathtt{S}_1, \ldots, \mathtt{S}_n)}\ [\text{S-TUP3}]$$

$$\frac{\forall \mathtt{i}.\mathtt{T}_i \leq \mathtt{S}}{\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n \leq \mathtt{S}}\ [\text{S-UNION1}] \qquad \frac{\exists \mathtt{i}.\mathtt{T} \leq \mathtt{S}_i}{\mathtt{T} \leq \mathtt{S}_1 \vee \ldots \vee \mathtt{S}_n}\ [\text{S-UNION2}]$$

$$\frac{\exists \mathtt{i}.\mathtt{T}_i \leq \mathtt{S}}{\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n \leq \mathtt{S}}\ [\text{S-INTERSECT1}] \qquad \frac{\forall \mathtt{i}.\mathtt{T} \leq \mathtt{S}_i}{\mathtt{T} \leq \mathtt{S}_1 \wedge \ldots \wedge \mathtt{S}_n}\ [\text{S-INTERSECT2}]$$

Fig. 3. A sound but *incomplete* subtyping relation for the language of types defined in §2.1

they make the subtyping problem harder (in fact, without tuples the subtyping problem for this system is fairly trivial).

2.4 Subtype Algorithm

Figure 2 employs an operation on types whose implementation is not immediately obvious — namely, determining whether one type subtypes another (i.e. $\mathtt{T}_1 \leq \mathtt{T}_2$). Indeed, there are many possible implementations of this operator and it is useful to consider two desirable properties:

Definition 2 (Subtype Soundness). *A subtype operator, $\leq$, is* sound *if, for any types $\mathtt{T}_1$ and $\mathtt{T}_2$, it holds that $\mathtt{T}_1 \leq \mathtt{T}_2 \implies [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$.*

Definition 3 (Subtype Completeness). *A subtype operator, $\leq$, is* complete *if, for any types $\mathtt{T}_1$ and $\mathtt{T}_2$, it holds that $[\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!] \implies \mathtt{T}_1 \leq \mathtt{T}_2$.*

A subtype operator which exhibits both of these properties is said to be *sound* and *complete*. We know of no previous flow typing system which has this property. The most notable existing system is that of Tobin-Hochstadt and Felleisen, who developed a flow type system for Racket (formerly PLT Scheme) [23, 24]. Like the system presented here, this supports subtyping in the presence of unions and negations and was shown to be sound. However, subtyping in their system is not complete, meaning that many potentially typeable programs cannot be typed.

Example. Figure 3 provides a typical set of rules defining a subtype relation over the language of types from §2.1. These have a natural recursive implementation and are comparable to those of [23, 24]. Rules S-ANY1, S-ANY2, S-INT1, S-INT2 and S-TUP1 are straightforward. We illustrate the remainder by example. Under S-TUP2, we have e.g. $(\texttt{int}, \texttt{int}) \leq \neg(\texttt{int}, \texttt{int}, \texttt{int})$ and $((\texttt{int}, \texttt{int}), \texttt{int}) \leq \neg(\texttt{int}, \texttt{int})$ whilst, similarly, we have $\neg(\texttt{any}, \texttt{any}) \leq \neg(\texttt{int}, \texttt{int})$ under S-TUP3. Under S-UNION1 we have e.g. $(\texttt{int}, \texttt{any}) \vee (\texttt{any}, \texttt{int}) \leq (\texttt{any}, \texttt{any})$ and e.g. $(\texttt{int}, \texttt{int}) \leq \texttt{int} \vee (\texttt{int}, \texttt{any})$ under S-UNION2. Finally, under S-INTERSECT1 we have e.g. $\texttt{int} \wedge (\texttt{int}, \texttt{int}) \leq \texttt{int}$ and e.g. $(\texttt{int}, \texttt{int}) \leq (\texttt{any}, \texttt{int}) \wedge (\texttt{int}, \texttt{any})$ under S-INTERSECT2.
The rules of Figure 3 can be shown as sound with respect to our semantic interpretation of types (i.e. Definition 1). However, they are evidently not complete. For example, neither of the following can be shown under Figure 3 (but are implied by Definition 1):

$$\texttt{any} \leq \texttt{int} \vee \neg\texttt{int}$$

$$(\texttt{int} \vee (\texttt{int}, \texttt{int}), \texttt{int}) \leq (\texttt{int}, \texttt{int}) \vee ((\texttt{int}, \texttt{int}), \texttt{int})$$

The rules of Figure 3 can be further extended to handle specific cases (such as those above). For example, we could add the following rules:

$$\frac{}{\texttt{any} \leq \texttt{T} \vee \neg\texttt{T}} \quad [\text{S-ANY3}]$$

$$\frac{\texttt{T} = (\texttt{T}_1, \ldots, \texttt{T}_{k-1}, \bullet, \texttt{T}_{k+1}, \texttt{T}_n)}{\texttt{T}[\bullet \mapsto \texttt{S}_1 \vee \ldots \vee \texttt{S}_n] \leq \texttt{T}[\bullet \mapsto \texttt{S}_1] \vee \ldots \vee \texttt{T}[\bullet \mapsto \texttt{S}_n]} \quad [\text{S-TUP4}]$$

S-ANY3 allows $\texttt{any} \leq \texttt{int} \vee \neg\texttt{int}$ to be shown, while S-TUP4 captures distributivity across tuples, allowing $(\texttt{int} \vee (\texttt{int}, \texttt{int}), \texttt{int}) \leq (\texttt{int}, \texttt{int}) \vee ((\texttt{int}, \texttt{int}), \texttt{int})$. However, adding additional rules seems (in our experience) somewhat futile and just forces ever-more esoteric counter-examples. For example, using the above rules we still cannot show $\texttt{any} \leq (\texttt{int}, \texttt{int}) \vee (\texttt{int}, \texttt{int}, \texttt{int}) \vee \big(\neg(\texttt{int}, \texttt{int}) \wedge \neg(\texttt{int}, \texttt{int}, \texttt{int})\big)$ (which is implied by Definition 1). In essence, the issue is that the number and variety of possible equivalences between types make it very difficult to construct a set of complete rules. To address this, our approach first normalises types to eliminate many such equivalences, and to make them more manageable.

2.5 Problem Statement

We can now succinctly express the problem addressed in this paper, namely: *to develop a sound and complete subtype algorithm for the language of types defined in §2.1.* We know of no previous algorithm with this property for a comparable language of types. In §4, we present such an algorithm which, in the worst case, requires an exponential number of steps to answer a subtyping query (in the size of the types involved). This complements the work of Frisch *et al.* who provided an existence proof but did not present a practical algorithm [33]. Furthermore, determining whether a polynomial time subtyping algorithm exists for this system remains, to the best of our knowledge, an open problem.

Positive Subtyping:

$$\frac{}{\mathtt{T}^+ \leq \mathtt{T}^+} \quad \text{(S-REFLEX)}$$

$$\frac{}{\mathtt{T}^+ \leq \mathtt{any}} \quad \text{(S-ANY)}$$

$$\frac{\forall \mathtt{i} \in \{1,\ldots,\mathtt{n}\}.\mathtt{T}_\mathtt{i}^+ \leq \mathtt{S}_\mathtt{i}^+}{(\mathtt{T}_1^+ \ldots,\mathtt{T}_\mathtt{n}^+) \leq (\mathtt{S}_1^+,\ldots,\mathtt{S}_\mathtt{n}^+)} \quad \text{(S-TUP)}$$

Fig. 4. Subtyping rules for positive atoms in FW

3 Preliminaries

Before we present our algorithm for sound and complete subtyping over the language of types defined in §2.1, we first introduce the key concepts which underpin it. These then form the building blocks for our algorithmic developments in the following section.

3.1 Atoms

An important aspect of our algorithm is the definition of an *atom*. These are indivisible types which are split into the *positive* and *negative* atoms as follows:

Definition 4 (Type Atoms). *Let* $\mathtt{T}^*$ *denote a* type atom*, defined as follows:*

$$\begin{array}{l}\mathtt{T}^* ::= \mathtt{T}^+ \mid \mathtt{T}^- \\ \mathtt{T}^- ::= \neg\mathtt{T}^+ \\ \mathtt{T}^+ ::= \mathtt{any} \mid \mathtt{int} \mid (\mathtt{T}_1^+,\ldots,\mathtt{T}_\mathtt{n}^+)\end{array}$$

Here, $\mathtt{T}^+$ *denotes a* positive atom *whilst* $\mathtt{T}^-$ *denotes a* negative atom.

We can see from Definition 4 that a negative atom is simply a negated positive atom. Furthermore, the elements of tuple atoms are themselves positive atoms — which differs from the original definition of types, where an element could hold any possible type (including e.g. a union or intersection type). As we will see, one of the challenges we face lies in the process of converting from the general types of §2.1 into the more restricted forms used here. For example, $(\mathtt{int} \vee (\mathtt{int},\mathtt{int}),\mathtt{any})$ can be converted into $(\mathtt{int},\mathtt{any}) \vee ((\mathtt{int},\mathtt{int}),\mathtt{any})$ — which is a union of positive atoms.

The first building block we require is that of subtyping between atoms. For our purposes, this operation need only be defined for positive atoms (a fact which at first surprised us), but could be extended to negative atoms as well. Figure 4 presents the subtyping relation between positive atoms. These employ judgements of the form "$\mathtt{T}_1 \leq \mathtt{T}_2$", which are read simply as: *the set of values described by* $\mathtt{T}_1$ *subtypes those of* $\mathtt{T}_2$. The rules of Figure 4 are mostly straightforward. Furthermore, we can trivially obtain soundness and completeness for the subtype relation given in Figure 4:

Lemma 1. *Let* $\mathtt{T}_1^+$ *and* $\mathtt{T}_2^+$ *be positive atoms. Then,* $\mathtt{T}_1^+ \leq \mathtt{T}_2^+ \iff [\![\mathtt{T}_1^+]\!] \subseteq [\![\mathtt{T}_2^+]\!]$.

Proof. Straightforward by inspection of Definition 1 and Figure 4. □

The second important building block is the observation that type atoms are *finitely indivisible*. That is, a type atom cannot be represented equivalently as a finite set of atoms which does not include itself:

Lemma 2 (Atom Indivisibility). *Let* $\mathtt{T}^+$ *and* $\mathtt{S}_1^+, \ldots, \mathtt{S}_\mathtt{n}^+$ *be positive type atoms where* $[\![\mathtt{T}^+]\!] = [\![\mathtt{S}_1^+ \cup \ldots \cup \mathtt{S}_\mathtt{n}^+]\!]$. *Then, for some* $\mathtt{1} \leq \mathtt{i} \leq \mathtt{n}$, *we have* $\mathtt{T}^+ = \mathtt{S}_\mathtt{i}^+$.

Proof. Straightforward by inspection of Definitions 1 + 4. □

The implications of Lemma 2 should not be overlooked. By construction, we have that $\forall \mathtt{i}.[\![\mathtt{S}_\mathtt{i}^+]\!] \subseteq [\![\mathtt{T}^+]\!]$ and, hence, $\mathtt{T}^+$ is the unique canonical representative of the set it describes (i.e. $[\![\mathtt{T}^+]\!]$). Furthermore, given any $\mathtt{S}_1^+, \ldots, \mathtt{S}_\mathtt{n}^+$ where $[\![\mathtt{T}^+]\!] = [\![\mathtt{S}_1^+ \cup \ldots \cup \mathtt{S}_\mathtt{n}^+]\!]$, we can quickly and easily identify $\mathtt{T}^+$ using the subtype operator of Figure 4.

The third important building block we require is that of (positive) atom intersection. We let $\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+$ denote the construction of a type representing the intersection of the values in $\mathtt{T}_1^+$ with those of $\mathtt{T}_2^+$. Note that $\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+$ produces either a positive atom or `void` (in the case of no intersection):

Definition 5 (Atom Intersection). *Let* $\mathtt{T}_1^+$ *and* $\mathtt{T}_2^+$ *be positive atoms. Then,* $\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+$ *is a positive atom or* `void` *determined as follows:*

$$
\begin{array}{lll}
\mathtt{T}^+ \sqcap \mathtt{T}^+ & = \mathtt{T}^+ & (1)\\
\mathtt{any} \sqcap \mathtt{T}^+ & = \mathtt{T}^+ & (2)\\
\mathtt{T}^+ \sqcap \mathtt{any} & = \mathtt{T}^+ & (3)\\
\mathtt{int} \sqcap (\mathtt{T}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+) & = \mathtt{void} & (4)\\
(\mathtt{T}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+) \sqcap \mathtt{int} & = \mathtt{void} & (5)\\
(\mathtt{T}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+) \sqcap (\mathtt{S}_1^+, \ldots, \mathtt{S}_\mathtt{m}^+) & = \mathtt{void}, \textbf{if } \mathtt{n} \neq \mathtt{m} \textbf{ or } \exists \mathtt{i}.\mathtt{T}_\mathtt{i}^+ \sqcap \mathtt{S}_\mathtt{i}^+ = \mathtt{void} & (6)\\
& = (\mathtt{T}_1^+ \sqcap \mathtt{S}_1^+, \ldots, \mathtt{T}_\mathtt{n}^+ \sqcap \mathtt{S}_\mathtt{n}^+), \textbf{otherwise} & (7)
\end{array}
$$

Observe that (2) + (3) and (4) + (5) are symmetric.

Definition 5 is straightforward. For example, $(\mathtt{int}) \sqcap (\mathtt{int}, \mathtt{int}) = \mathtt{void}$ as the number of fields differs (i.e. following Definition 1 where $[\![(\mathtt{int})]\!] \cap [\![(\mathtt{int}, \mathtt{int})]\!] = \emptyset$). Also, $(\mathtt{any}, \mathtt{any}) \sqcap (\mathtt{int}, \mathtt{int}) = (\mathtt{int}, \mathtt{int})$ as expected. Note, $\mathtt{int} \sqcap (\mathtt{int}) = \mathtt{void}$ since $(\mathtt{T})$ is a tuple of arity-1 and is considered distinct from `T`. Finally, we can trivially obtain soundness and completeness for this operation:

Lemma 3. *Let* $\mathtt{T}_1^+$ *and* $\mathtt{T}_2^+$ *be atoms. Then,* $[\![\mathtt{T}_1^+ \sqcap \mathtt{T}_2^+]\!] = [\![\mathtt{T}_1^+]\!] \cap [\![\mathtt{T}_2^+]\!]$.

Proof. Straightforward by inspection of Definition 1 and Definition 5. □

3.2 Disjunctive Normal Form (DNF)

We now consider the procedure for converting a general type into a more classical Disjunctive Normal Form (DNF):

Definition 6 (DNF). *Let* $\mathtt{T} \Longrightarrow^* \mathtt{T}'$ *denote the application of zero or more rewrite rules (defined below) to type* $\mathtt{T}$*, producing a potentially updated type* $\mathtt{T}'$*.*

$$
\begin{array}{lll}
\neg\neg \mathtt{T} & \Longrightarrow \mathtt{T} & (1)\\
\neg \bigvee_i \mathtt{T_i} & \Longrightarrow \bigwedge_i \neg \mathtt{T_i} & (2)\\
\neg \bigwedge_i \mathtt{T_i} & \Longrightarrow \bigvee_i \neg \mathtt{T_i} & (3)\\
\left(\bigvee_i \mathtt{S_i}\right) \wedge \bigwedge_j \mathtt{T_j} & \Longrightarrow \bigvee_i \left(\mathtt{S_i} \wedge \bigwedge_j \mathtt{T_j}\right) & (4)\\
(\ldots, \bigvee_i \mathtt{T_i}, \ldots) & \Longrightarrow \bigvee_i (\ldots, \mathtt{T_i}, \ldots) & (5)\\
(\ldots, \bigwedge_i \mathtt{T_i}, \ldots) & \Longrightarrow \bigwedge_i (\ldots, \mathtt{T_i}, \ldots) & (6)\\
(\ldots, \neg \mathtt{T}, \ldots) & \Longrightarrow (\ldots, \mathtt{any}, \ldots) \wedge \neg(\ldots, \mathtt{T}, \ldots) & (7)
\end{array}
$$

$\mathtt{DNF}(\mathtt{T}) = \mathtt{T}'$ *denotes the computation* $\mathtt{T} \Longrightarrow^* \mathtt{T}'$*, such that no more rewrite rules apply.*

Here, $\bigvee_i \mathtt{T_i}$ (resp. $\bigwedge_i \mathtt{T_i}$) represents a finite disjunction of the form $\mathtt{T_1} \vee \ldots \vee \mathtt{T_n}$ (resp. $\mathtt{T_1} \wedge \ldots \wedge \mathtt{T_n}$). The above rules convert a type into something similar to the classical notion of disjunctive normal form for logical expressions, with the key difference being that we must additionally factor unions, intersections and negations out of tuples. Rules 2+3 push negations inwards such that, for example, $\neg(\mathtt{T_1} \wedge \mathtt{T_2})$ rewrites to $\neg\mathtt{T_1} \vee \neg\mathtt{T_2}$. Rule 4 factors unions out of intersections, such that e.g. $\mathtt{T_1} \wedge (\mathtt{T_2} \vee \mathtt{T_3})$ rewrites to $(\mathtt{T_1} \wedge \mathtt{T_2}) \vee (\mathtt{T_1} \wedge \mathtt{T_3})$. Recall from §2.1 that $\mathtt{T_1} \wedge \mathtt{T_2}$ is indistinguishable from $\mathtt{T_2} \wedge \mathtt{T_1}$ and, hence, rule 4 is not restricted to rewriting only a leftmost union (as the presentation might suggest). Rules 5, 6 + 7 are responsible for factoring union and intersection types out of tuples. For example, $(\mathtt{int} \vee (\mathtt{int}, \mathtt{int}), \mathtt{any})$ rewrites by rule 4 to $(\mathtt{int}, \mathtt{any}) \vee ((\mathtt{int}, \mathtt{int}), \mathtt{any})$. Similarly, $(\mathtt{any} \wedge \neg\mathtt{int}, \mathtt{any})$ rewrites by rule 6 and then by rule 7 to give $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any})$. Finally, we note that $\mathtt{DNF}(\mathtt{T})$ may produce an exponential number of terms in the worst-case [38–40].

We now list several properties which can be trivially shown for the function $\mathtt{DNF}(\mathtt{T})$, and more details can be found in [41]:

Lemma 4 (DNF Construction). *Let* $\mathtt{T}$ *be a type where* $\mathtt{DNF}(\mathtt{T}) = \mathtt{T}'$*. Then,* $\mathtt{T}'$ *has the form* $\bigvee_i \bigwedge_j \mathtt{T}^*_{i,j}$*.*

Lemma 5 (DNF Preservation). *Let* $\mathtt{T}$ *be a type where* $\mathtt{T} \Longrightarrow \mathtt{T}'$ *by a rewrite rule from Definition 6. Then,* $[\![\mathtt{T}]\!] = [\![\mathtt{T}']\!]$*.*

Lemma 6 (DNF Termination). *Let* $\mathtt{T}$ *be a type. Then, there exists a type* $\mathtt{T}'$ *for which no further rewrite rules from Definition 6 apply, such that* $\mathtt{T} \Longrightarrow^* \mathtt{T}'$*.*

In considering Lemma 4, recall from Definition 4 that a type $\mathtt{T}^*$ represents a positive or negative atom. Thus, $\mathtt{T}^*$ is either a positive atom or a negated positive atom and may recursively contain only positive atoms.

4 Subtyping Algorithm

We now present our algorithm for sound and complete subtyping over the language of types defined in §2.1. We begin with an overview of the problem and our solution, and then proceed to progressively introduce the main pieces of the algorithm.

4.1 Overview

Let us reconsider the example subtyping algorithm presented in Figure 3. Recall that, whilst this algorithm can be shown sound, it is not complete. In particular, the following two rules are problematic:

$$\frac{\forall \mathtt{i}.\mathtt{T_i} \leq \mathtt{S}}{\mathtt{T_1} \vee \ldots \vee \mathtt{T_n} \leq \mathtt{S}}\ [\text{S-UNION1}] \qquad \frac{\exists \mathtt{i}.\mathtt{T} \leq \mathtt{S_i}}{\mathtt{T} \leq \mathtt{S_1} \vee \ldots \vee \mathtt{S_n}}\ [\text{S-UNION2}]$$

The problem is that examples of the form $\mathtt{T_1} \vee \mathtt{T_2} \leq \mathtt{T_3} \vee \mathtt{T_4}$ where $[\![\mathtt{T_1} \vee \mathtt{T_2}]\!] \subseteq [\![\mathtt{T_3} \vee \mathtt{T_4}]\!]$ exist, but where neither S-UNION1 nor S-UNION2 can apply (and, hence, such examples cannot be shown under Figure 3). The following illustrates two such examples:

$$\mathtt{int} \vee \neg\mathtt{int} \leq (\mathtt{int}, \mathtt{int}) \vee \neg(\mathtt{int}, \mathtt{int}) \tag{1}$$

$$((\mathtt{int}, \mathtt{int}), \mathtt{any}) \leq ((\mathtt{int}, \mathtt{int}), \mathtt{int}) \vee \neg(\mathtt{any}, \mathtt{int}) \tag{2}$$

Another example is $(\mathtt{int}, \mathtt{any}) \wedge (\mathtt{any}, \mathtt{int}) \leq (\mathtt{int}, \mathtt{int})$ which exploits a similar problem with the S-INTERSECT1 rule.

The problem common to all these examples seems to be the number and variety of equivalences between types. To tackle these problems, we build our algorithm around the intuition that $[\![\mathtt{T_1}]\!] \subseteq [\![\mathtt{T_2}]\!]$ iff $[\![\mathtt{T_1}]\!] - [\![\mathtt{T_2}]\!] = \emptyset$. This requires an algorithm for computing the difference of two types, such that $\mathtt{T_1} - \mathtt{T_2} = \mathtt{void}$ iff $[\![\mathtt{T_1}]\!] - [\![\mathtt{T_2}]\!] = \emptyset$. When types are represented as disjuncts of canonical conjuncts (referred to as *Canonicalised Disjunctive Normal Form* or DNF$^+$ for short), then computing their difference in a way that obtains the desired property is relatively easy. We proceed by first defining the notion of a *canonical conjunct* (§4.2) and then how one is constructed (§4.3). Finally, we show how a general type can be converted into DNF$^+$ (§4.4), and put the whole thing together illustrated with an example (§4.5).

4.2 Canonical Conjuncts

The first step in the canonicalisation process is to canonicalise intersections of the form $\mathtt{T_1} \wedge \ldots \wedge \mathtt{T_n}$. For example, $\mathtt{int} \wedge \mathtt{any}$ can be safely simplified to $\mathtt{int}$. Likewise, $(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{any})$ can be simplified to $\mathtt{void}$, while $(\mathtt{int}, \mathtt{int}) \wedge \neg\mathtt{int}$ can be simplified to $(\mathtt{int}, \mathtt{int})$ and, finally, $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any})$ can be simplified to $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$. As we will see, any intersection $\bigwedge_\mathtt{i} \mathtt{T_i^*}$ between atoms can be represented as a positive atom conjuncted with zero or more negative atoms, i.e. as $\mathtt{T_1^+} \wedge \neg\mathtt{T_2^+} \wedge \ldots \wedge \neg\mathtt{T_n^+}$.

Given the tools developed in §3.1 (i.e. Figure 4 and Definition 5), we can now formalise the notion of a *canonical conjunct* as follows:

Definition 7 (Canonical Conjunct). *Let* $\mathtt{T}^\wedge$ *denote a* canonical conjunct. *Then,* $\mathtt{T}^\wedge$ *is a type of the form* $\mathtt{T}_1^+ \wedge \neg\mathtt{T}_2^+ \wedge \ldots \wedge \neg\mathtt{T}_n^+$ *where:*

1. *For every negation* $\neg\mathtt{T}_k^+$*, we have* $\mathtt{T}_1^+ \not\equiv \mathtt{T}_k^+$ *and* $\mathtt{T}_1^+ \geq \mathtt{T}_k^+$.
2. *For any two distinct negations* $\neg\mathtt{T}_k^+$ *and* $\neg\mathtt{T}_m^+$*, we have* $\mathtt{T}_k^+ \not\geq \mathtt{T}_m^+$.

Recall the subtype relation, $\geq$, between positive atoms used in Definition 7 is given in Figure 4. Now, rule 1 makes sense if we recall that $\mathtt{T}_1 \wedge \neg\mathtt{T}_2$ can be thought of as $\mathtt{T}_1 - \mathtt{T}_2$; thus, in rule 1 we require that the amount "subtracted" from the positive atom by any given negative atom is strictly less than the total. For example, $(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{any})$ is not permitted as this corresponds to the `void` (i.e. empty) type. Likewise, $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any})$ is not permitted either since this is more precisely represented as $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$. Rule 2 prohibits negative atoms from subsuming each other. For example, $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \wedge \neg(\mathtt{any}, \mathtt{int})$ is more precisely represented as $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{any}, \mathtt{int})$. Note, we need not worry about atoms that overlap but where neither subsumes the other (i.e. where $[\![\mathtt{T}_1^+]\!] \cap [\![\mathtt{T}_2^+]\!] \neq \emptyset$ but $[\![\mathtt{T}_1^+]\!] \not\subseteq [\![\mathtt{T}_2^+]\!]$ and $[\![\mathtt{T}_1^+]\!] \not\supseteq [\![\mathtt{T}_2^+]\!]$) — this follows from Lemma 2 (indivisibility) as such types canonically represent distinct sets (hence must be retained in the conjunct).

We can make the following strong statement about canonical conjuncts based on Definition 7 — namely, that canonical conjuncts are indeed canonical:

Lemma 7 (Canonical Conjuncts). *Let* $\mathtt{T}^\wedge = \mathtt{T}_1^+ \wedge \bigwedge_i \neg\mathtt{T}_i^+$ *and* $\mathtt{S}^\wedge = \mathtt{S}_1^+ \wedge \bigwedge_j \neg\mathtt{S}_j^+$ *be canonical conjuncts. Then, it follows that* $[\![\mathtt{T}^\wedge]\!] = [\![\mathtt{S}^\wedge]\!] \iff \mathtt{T}^\wedge = \mathtt{S}^\wedge$.

Proof. Follows from proof of atom indivisibility (Lemma 2). See [41] for details. □

4.3 Conjunct Construction

We now develop the mechanism for constructing a canonical conjunct from an arbitrary conjunct of atoms:

Definition 8 (Conjunct Canonicalisation). *Let* $\bigwedge_i \mathtt{T}_i^* \Longrightarrow^* \bigwedge_j \mathtt{S}_j^*$ *denote the application of zero or more rewrite rules (defined below) to* $\bigwedge_i \mathtt{T}_i^*$*, producing a potentially updated version* $\bigwedge_j \mathtt{S}_j^*$.

$$
\begin{array}{lcll}
\mathtt{void} \wedge \ldots & \Longrightarrow & \mathtt{void} & & (1)\\
\mathtt{T}_i^+ \wedge \mathtt{T}_j^+ \wedge \ldots & \Longrightarrow & (\mathtt{T}_i^+ \sqcap \mathtt{T}_j^+) \wedge \ldots & & (2)\\
\mathtt{T}_x^+ \wedge \neg\mathtt{T}_y^+ \wedge \ldots & \Longrightarrow & \mathtt{void} & \textbf{if } \mathtt{T}_x^+ \leq \mathtt{T}_y^+ & (3)\\
 & \Longrightarrow & \mathtt{T}_x^+ \wedge \ldots & \textbf{if } \mathtt{T}_x^+ \sqcap \mathtt{T}_y^+ = \mathtt{void} & (4)\\
 & \Longrightarrow & \mathtt{T}_x^+ \wedge \neg(\mathtt{T}_x^+ \sqcap \mathtt{T}_y^+) \wedge \ldots & \textbf{if } \mathtt{T}_x^+ \not\geq \mathtt{T}_y^+ & (5)\\
\neg\mathtt{T}_x^+ \wedge \neg\mathtt{T}_y^+ \wedge \ldots & \Longrightarrow & \neg\mathtt{T}_x^+ \wedge \ldots & \textbf{if } \mathtt{T}_x^+ \geq \mathtt{T}_y^+ & (6)
\end{array}
$$

Let $\mathtt{CAN}(\bigwedge_i \mathtt{T}_i^*) = \bigwedge_j \mathtt{S}_j^*$ *denote the computation* $\bigwedge_i \mathtt{T}_i^* \Longrightarrow^* \bigwedge_j \mathtt{S}_j^*$*, such that no further rewrite rules apply.*

In considering the rules from Definition 8, we must recall that $\mathtt{T}_1 \wedge \mathtt{T}_2$ is not distinguishable from $\mathtt{T}_2 \wedge \mathtt{T}_1$. Therefore e.g. rule (2) picks two arbitrary positive atoms from $\bigwedge_i \mathtt{T}_i^*$, not just the leftmost two (as the presentation might suggest). Rule 2 simply combines all the positive atoms together using the intersection operator for positive atoms (Definition 5). After repeated applications of rule 2, there will be at most one positive atom remaining. Rule 3 catches the case when the negative contribution exceeds the positive contribution (e.g. $\mathtt{int} \wedge \neg\mathtt{any} \implies \mathtt{void}$). Rule 4 catches negative components which lie outside the domain (e.g. $\mathtt{int} \wedge \neg(\mathtt{int}, \mathtt{int}) \implies \mathtt{int}$). Rule 5 covers the case of negative components which lie partially outside the domain (e.g. $(\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{any}) \implies (\mathtt{any}, \mathtt{int}) \wedge \neg(\mathtt{int}, \mathtt{int})$). Finally, rule 6 catches the case where one negative component is completely consumed by another (e.g. $(\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{int}) \implies (\mathtt{any}, \mathtt{any}) \wedge \neg(\mathtt{int}, \mathtt{any})$).

Lemma 8. *Let $\bigwedge_i \mathtt{T}_i^*$ be an arbitrary conjunct of atoms containing at least one positive atom. Then, $\mathtt{CAN}(\bigwedge_i \mathtt{T}_i^*)$ is either a canonical conjunct or* `void`.

Proof. Straightforward by case analysis on the ways in which an arbitrary conjunct of atoms does not meet the requirements of Definition 7 (and, for each case, that a rule from Definition 8 applies). See [41] for details. □

The requirement in Lemma 8 for at least one positive atom arises because the rules of Definition 8 do not introduce positive atoms, but canonical conjuncts require them. In fact, we can easily ensure an arbitrary conjunct of atoms, $\bigwedge_i \mathtt{T}_i^*$, has at least one positive atom — we simply add `any` to give $\mathtt{any} \wedge \bigwedge_i \mathtt{T}_i^*$.

Definition 9 (Conjunct Intersection). *Let $\mathtt{T}_1^\wedge, \ldots, \mathtt{T}_n^\wedge$ be canonical conjuncts. Then, $\mathtt{T}_1^\wedge \sqcap \ldots \sqcap \mathtt{T}_n^\wedge$ denotes their intersection, and is defined as $\mathtt{CAN}(\mathtt{T}_1^\wedge \wedge \ldots \wedge \mathtt{T}_n^\wedge)$.*

Observe that, by construction, $\bigsqcap_i \mathtt{T}_i^\wedge$ yields either a canonical conjunct or `void`. Since a canonical conjunct cannot represent `void`, we have the required property that $[\![\bigwedge_i \mathtt{T}_i^\wedge]\!] = \emptyset \iff \bigsqcap_i \mathtt{T}_i^\wedge = \mathtt{void}$. To see why a canonical conjunct cannot represent `void`, recall that `void` is short-hand for $\neg\mathtt{any}$. Thus, we might consider $\mathtt{any} \wedge \neg\mathtt{any}$ to be a canonical conjunct representing `void` — but, this is invalid as, for a type $\mathtt{T}_1 \wedge \neg\mathtt{T}_2$ to be a canonical conjunct, Definition 7 requires $\mathtt{T}_1 > \mathtt{T}_2$. In fact, by construction, no canonical conjunct $\mathtt{T}^\wedge$ exists where $[\![\mathtt{T}^\wedge]\!] = [\![\mathtt{void}]\!]$. Finally, the following ensures the overall canonicalisation process is sound:

Lemma 9. *Let $\mathtt{T}_1^\wedge, \ldots, \mathtt{T}_n^\wedge$ be canonical conjuncts. Then, $[\![\bigwedge_i \mathtt{T}_i^\wedge]\!] = [\![\bigsqcap_i \mathtt{T}_i^\wedge]\!]$.*

Proof. Straightforward by case analysis on the rules of Definition 8 to show that each rule preserves the described semantic set before and after the rewrite. See [41] for details. □

4.4 Canonicalised Disjunctive Normal Form (DNF$^+$)

Finally, we can now formally define the process for converting an arbitrary type (as defined in §2.1) into the variant of disjunctive normal form we refer to as *Canonicalised Disjunctive Normal Form* (DNF$^+$):

Definition 10 (DNF$^+$). *Let* $\mathtt{T}^\vee$ *denote a type in* Canonicalised Disjunctive Normal Form (DNF$^+$). *Then, either* $\mathtt{T}^\vee$ *has the form* $\bigvee_i \mathtt{T}_i^\wedge$ *or is* `void`.

In our definition of DNF$^+$, we must include a special case for when $\mathtt{T} = \mathtt{void}$ since (as discussed earlier) `void` is not a canonical conjunct. We can now easily construct types in DNF$^+$ as follows:

Definition 11 (DNF$^+$ Construction). *Let* $\mathtt{T}$ *be a type where* $\mathtt{T}' = \mathtt{DNF}(\mathtt{T})$ *and, hence, by Lemma 4 we have* $\mathtt{T}' = \bigvee_i \bigwedge_j \mathtt{T}^*_{i,j}$. *Then,* $\mathtt{DNF}^+(\mathtt{T}) = \bigvee_i \bigcap_j \mathtt{T}^*_{i,j}$.

In considering Definition 11, we must recall our assumption from §2.1 that $\mathtt{T}_1 \vee \mathtt{T}_1$ is indistinguishable from $\mathtt{T}_1$. This is important as it ensures that, if all the intersected conjuncts give `void`, then the overall result is `void` (i.e. since $\mathtt{void} \vee \mathtt{void} = \mathtt{void}$, etc). This reflects our overall goal of ensuring $[\![\mathtt{T}]\!] = \emptyset \iff \mathtt{DNF}^+(\mathtt{T}) = \mathtt{void}$. We now present the overall theorem of this paper:

Theorem 1. *Let* $\mathtt{T}$ *be a type (as defined in §2.1). Then,* $[\![\mathtt{T}]\!] = [\![\mathtt{DNF}^+(\mathtt{T})]\!]$.

Proof. Follows immediately from Lemma 9 and Definition 11. □

4.5 Putting It All Together

We can now give a formal definition for our subtyping operator which is sound and complete for the language of types defined in §2.1, and which replaces Figure 3:

Definition 12 (Subtyping). *Let* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *be types (as defined in §2.1). Then, it follows that* $\mathtt{T}_1 \leq \mathtt{T}_2 \iff \mathtt{DNF}^+(\mathtt{T}_1 \wedge \neg \mathtt{T}_2) = \mathtt{void}$.

The proof that this rule is sound and complete follows immediately from Theorem 1. Furthermore it is important to realise that, whilst Definition 12 offers an improvement over Figure 3 in terms of completeness, this comes at a cost. More specifically, the subtype relation defined in Figure 3 can be implemented with a polynomial time algorithm, whilst our replacement (i.e. Definition 12) requires exponential time in the worst case. This is because the first step of the process which converts $\mathtt{T}$ into disjunctive normal form (i.e. Definition 6) can produce an exponential explosion in the number of terms [38–40]. As such, determining whether a sound and complete polynomial time subtyping algorithm exists remains an open problem.

We now return to consider a simple example from §2.4, and illustrate how it is resolved:

$$\mathtt{any} \leq \mathtt{int} \vee \neg\mathtt{int} \iff \mathtt{DNF}^+(\mathtt{any} \wedge \neg(\mathtt{int} \vee \neg\mathtt{int})) = \mathtt{void}$$

$$\begin{aligned} \mathtt{DNF}^+(\mathtt{any} \wedge \neg(\mathtt{int} \vee \neg\mathtt{int})) &= \mathtt{CAN}(\mathtt{DNF}(\mathtt{any} \wedge \neg(\mathtt{int} \vee \neg\mathtt{int}))) \\ &= \mathtt{CAN}(\mathtt{any} \wedge \neg\mathtt{int} \wedge \mathtt{int}) \\ &= \mathtt{void} \end{aligned}$$

Therefore, the algorithm correctly concludes that $\mathtt{any} \leq \mathtt{int} \vee \neg\mathtt{int}$ holds.

5 Related Work

Tobin-Hochstadt and Felleisen consider the problem of typing previously untyped Racket (aka Scheme) programs using a flow-typing algorithm [23, 24]. Their system retypes variables within expressions dominated by type tests. However, they employ only union types and do not consider intersections or negations, making their system significantly more conservative than presented here. The work of Guha *et al.* focuses on flow-sensitive type checking for JavaScript [25]. This assumes programmer annotations are given for parameters, and operates in two phases: first, a flow analysis inserts special runtime checks; second, a standard (i.e. flow-insensitive) type checker operates on the modified AST. Their system employs union types, but does not support negations or intersections. Furthermore, it can retype variables as a result of runtime type tests, although only simple forms are permitted.

Since locals and stack locations are untyped in Java Bytecode, the Java Bytecode Verifier employs flow typing to ensure type safety [31]. The verifier retypes variables after assignments, but does not retype them after `instanceof` tests. And, instead of supporting explicit unions, it computes the least upper bound of the types for each variable at a meet point. A well-known problem, however, is that Java's subtype relation does not form a complete lattice [32]. This arises because two classes can share the same super-class and implement the same interface; thus, they may not have a unique least upper bound. The solution adopted by the bytecode verifier ignores interfaces entirely and, instead, maps them to `java.lang.Object`. This approach is conservative and means some programs will fail to verify that we might otherwise expect to pass. Several works on formalising the bytecode verifier have proposed the use of intersection types as an alternative solution [42, 43].

Type qualifiers constrain the possible values a variable may hold. CQual is a flow-sensitive qualifier inference supporting numerous type qualifiers, including those for synchronisation and file I/O [12]. CQual does not account for the effects of conditionals and, hence, retyping is impossible. Fähndrich and Leino discuss a system for checking non-null qualifiers in the context of C# [15]. Here, variables are annotated with `NonNull` to indicate they cannot hold **`null`**. Non-null qualifiers are interesting because they require variables be retyped after conditionals (i.e. retyping `v` from `Nullable` to `NonNull` after `v!=`**`null`**). Fähndrich and Leino hint at the use of retyping, but focus primarily on issues related to object constructors. Ekman *et al.* implemented this system within the JustAdd compiler, although few details are given regarding variable retyping [13]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `v!=`**`null`** checks [21]. The JACK tool for verifying `@NonNull` type annotations extends the bytecode verifier with an extra level of indirection called *type aliasing* [14]. This enables the system to retype a variable `x` as `@NonNull` in the body of an **`if`**`(x!=`**`null`**`)` conditional. The algorithm is formalised using a flow-sensitive type system operating on Java bytecode. JavaCOP provides an expressive language for writing type system extensions, including non-null types [22]. This system is flow-insensitive and cannot account for the effects of conditionals; as a work around, the tool allows assignment from a nullable variable `v` to a non-null variable if this is the first statement after a `v!=null` conditional.

Finally, there have been some attempts to incorporate intersection and union types into mainstream languages. The most relevant is that of Büchi and Weck who introduce *compound types* in to Java to overcome limitations caused by a lack of multiple inheritance [44]. Another interesting work is that of Igarashi and Nagira, who introduce union types into Java to increase code reusability [45]. Likewise, Plümicke introduces intersection types into Java to ensure methods have principle types [46]. This helps to alleviate the burden of writing complex types in some situations.

6 Conclusion

Flow-typing systems often require complex type systems involving unions, intersections and/or negations. For example, unions are often used to describe the types of variables at meet points. Likewise, intersections and negations can describe the effect of runtime type tests. However, subtype testing is a challenging algorithmic problem for a type system containing these features. In particular, to ensure the greatest number of programs as possible can be typed, we desire that subtyping is both *sound* and *complete*. Frisch *et al.* demonstrated that this problem was decidable [33]. However, their proof was not constructive and did not lend itself naturally to an implementation. In this paper, we presented a sound and complete algorithm for subtyping in the presence of unions, intersections and negations. This contrasts with previous flow type systems (e.g. [23, 24]) which are shown sound, but not complete.

We framed our algorithm in the context of a flow typing system, which is a natural fit for this work and has many well-known practical applications. Furthermore, our motivation for developing this algorithm stems from our work on the Whiley programming language [28–30], which incorporates an ambitious flow type system. However, there are other potential applications for our algorithm, such as e.g. typing XML Schema [47, 48].

Acknowledgements. The author would like to thank Sophia Drossopoulou and Nicholas Cameron for helpful comments on earlier drafts of this paper. This work is supported by the Marsden Fund, administered by the Royal Society of New Zealand.

References

1. Cartwright, R., Fagan, M.: Soft typing. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI), pp. 278–292 (1991)
2. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: Proceedings of the Dynamic Languages Symposium (DLS), pp. 53–64 (2007)
3. Ousterhout, J.K.: Scripting: Higher-level programming for the 21st century. IEEE Computer 31(3), 23–30 (1998)
4. Spinellis, D.: Java makes scripting languages irrelevant? IEEE Software 22(3), 70–71 (2005)
5. Loui, R.P.: In praise of scripting: Real programming pragmatism. IEEE Computer 41(7), 22–26 (2008)

6. Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strnisa, R., Vitek, J., Wrigstad, T.: Thorn: robust, concurrent, extensible scripting on the JVM. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 117–136 (2009)
7. Hindley, J.R.: The principal type-scheme of an object in combinatory logic. Transactions of the AMS 146, 29–60 (1969)
8. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)
9. The Scala programming language, http://lamp.epfl.ch/scala/
10. Bierman, G., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 479–498 (2007)
11. Remy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory And Practice of Object Systems 4(1), 27–50 (1998)
12. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI), pp. 1–12 (2002)
13. Ekman, T., Hedin, G.: Pluggable checking and inferencing of non-null types for Java. Journal of Object Technology 6(9), 455–475 (2007)
14. Male, C., Pearce, D.J., Potanin, A., Dymnikov, C.: Java bytecode verification for @NonNull types. In: Proceedings of the Confererence on Compiler Construction (CC), pp. 229–244 (2008)
15. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 302–312 (2003)
16. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL), pp. 228–241 (1999)
17. Hunt, S., Sands, D.: On flow-sensitive security types. In: Proceedings of the ACM Symposium on the Principles Of Programming Languages (POPL), pp. 79–90 (2006)
18. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proc. CSF, pp. 186–199 (2010)
19. Pearce, D.J.: JPure: A Modular Purity System for Java. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 104–123. Springer, Heidelberg (2011)
20. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI), pp. 192–203 (1999)
21. Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L., Verbrugge, C.: A Framework for Optimizing Java Using Attributes. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 334–554. Springer, Heidelberg (2001)
22. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 57–74 (2006)
23. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: Proceedings of the ACM Symposium on the Principles Of Programming Languages (POPL), pp. 395–406 (2008)
24. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: Proceedings of the ACM International Conference on Functional Programming (ICFP), pp. 117–128 (2010)
25. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing Local Control and State Using Flow Analysis. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 256–275. Springer, Heidelberg (2011)

26. Winther, J.: Guarded type promotion: eliminating redundant casts in Java. In: Proceedings of the Workshop on Formal Techniques for Java-like Programs, pp. 6:1–6:8 (2011)
27. What's new in Groovy 2.0?, http://www.infoq.com/articles/new-groovy-20
28. The Whiley programming language, http://whiley.org
29. Pearce, D., Noble, J.: Implementing a language with flow-sensitive and structural typing on the JVM. Electronic Notes in Computer Science 279(1), 47–59 (2011)
30. Pearce, D.J., Cameron, N., Noble, J.: Whiley: a language with flow-typing and updateable value semantics. Technical Report ECSTR12-09, Victoria University of Wellington (2012)
31. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley (1999)
32. Leroy, X.: Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning 30(3/4), 235–269 (2003)
33. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM 55(4), 19:1–19:64 (2008)
34. Aiken, A., Wimmers, E.L.: Type inclusion constraints and type inference. In: Proc. FPCA, pp. 31–41 (1993)
35. Damm, F.M.: Subtyping with Union Types, Intersection Types and Recursive Types. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 687–706. Springer, Heidelberg (1994)
36. Castagna, G., Frisch, A.: A Gentle Introduction to Semantic Subtyping. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 30–34. Springer, Heidelberg (2005)
37. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: Proceedings of the ACM/IEEE Symposium on Logic In Computer Science (LICS), pp. 137–146 (2002)
38. Garey, M.R., Johnson, D.S.: Computers and intractability; a guide to the theory of *NP*-completeness. W.H. Freeman (1979)
39. Umans, C.: The minimum equivalent DNF problem and shortest implicants. Journal of Computer and System Sciences 63 (2001)
40. Buchfuhrer, D., Umans, C.: The complexity of boolean formula minimization. Journal of Computer and System Sciences 77(1), 142–153 (2011)
41. Pearce, D.J.: Sound and complete flow typing with unions, intersections and negations. Technical Report ECSTR12-20, Victoria University of Wellington (2012)
42. Goldberg, A.: A specification of Java loading and bytecode verification. In: Proc. CCS, pp. 49–58 (1998)
43. Pusch, C.: Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 89–103. Springer, Heidelberg (1999)
44. Büchi, M., Weck, W.: Compound types for java. In: Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 362–373 (1998)
45. Igarashi, A., Nagira, H.: Union types for object-oriented programming. In: Proceedings of the Symposium on Applied Computing (SAC), pp. 1435–1441 (2006)
46. Plümicke, M.: Intersection types in java. In: Proceedings of the Conference on Principles and Practices of Programming in Java (PPPJ), pp. 181–188. ACM, New York (2008)
47. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)
48. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric general-purpose language. In: Proceedings of the ACM International Conference on Functional Programming (ICFP), pp. 51–63 (2003)

Knockout Prediction for Reaction Networks with Partial Kinetic Information

Mathias John[1,2], Mirabelle Nebut[1,2], and Joachim Niehren[1,3]

[1] BioComputing, Lifl (Cnrs Umr8022)
[2] University of Lille
[3] Inria Lille

Abstract. In synthetic biology, a common application field for computational methods is the prediction of knockout strategies for reaction networks. Thereby, the major challenge is the lack of information on reaction kinetics. In this paper, we propose an approach, based on abstract interpretation, to predict candidates for reaction knockouts, relying only on partial kinetic information. We consider the usual deterministic steady state semantics of reaction networks and a few general properties of reaction kinetics. We introduce a novel abstract domain over pairs of real domain values to compute the differences between steady states that are reached before and after applying some knockout. We show that this abstract domain allows us to predict correct knockout strategy candidates independent of any particular choice of reaction kinetics. Our predictions remain candidates, since our abstract interpretation over-approximates the solution space. We provide an operational semantics for our abstraction in terms of constraint satisfaction problems and illustrate our approach on a realistic network.

Keywords: Abstract interpretation, deterministic semantics, steady state, constraint satisfaction, synthetic biology.

1 Introduction

Synthetic biology aims at creating artificial micro-organisms, either by constructing them from scratch or by modifying existing once [2,1,21]. To this end, computational modeling is applied to predict the dynamic behavior of the resulting organisms [24,32]. Thereby, it is a common task to abstract micro-organisms as sets of chemical reactions and to predict the effects of reaction deletion on the dynamics of the concentrations of chemical species [38,5]. Such knockout considerations are in particular applied to predict strategies that increase the rate of some target reactions of interest.

Kinetic information is essential to predict the dynamics of chemical systems [22]. In reaction networks, each reaction is endowed with a kinetic rate law, that is a function that defines a rate in dependence of the concentration of the reaction's reactants. Common examples of rate laws are the law of mass action and Michaelis-Menten kinetics. These depend on kinetic rate constants

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 355–374, 2013.

that, however, remain mostly unknown. Moreover, in practice, many of a model's reactions combine several unknown reaction steps to one, see e.g. [19,20]. In these cases, only basic properties of the kinetics function are given, e.g. that the rate increases or decreases with the concentration of some chemical species. Thus, the precise prediction of the dynamics of chemical systems remains out of reach.

The commonly accepted dynamic semantics of systems of chemical reactions describes species concentrations in terms of stochastic processes [18]. When dealing with high concentrations, a meaningful approximation is to ignore the moments of probability distributions with an order greater than one [40,17]. Such, so called, deterministic semantics is then represented by systems of ordinary differential equations (ODEs) that describe the changes in the mean of species concentrations over time. Knockout predictions then usually regard fix points of ODEs when no more changes in concentrations occur (steady state). It is known that in nature systems of chemical reactions exist that have none or more than one steady state [37,11,14].

Abstract interpretation [8] has been introduced for the static analysis of program semantics. The idea is to approximate the state space of programs based on approximations of domains and computations. In the realm of chemical reaction systems, abstract interpretation has been applied to obtain different approximations of their dynamic, stochastic semantics [12,10] or to decrease the size of (infinitely) large sets of ODEs representing their deterministic semantics [6,13].

In this paper we propose an approach based on abstract interpretation, that predicts candidates for reaction knockout, with only partial kinetic information. We focus on cases that are on one hand simple, since reaction systems are assumed to always reach a steady state as it is usually done when predicting reaction knockouts [39,5]. On the other hand, they are complicated, since kinetics functions are completely unknown, except for a few basic properties. Our major idea is to compare steady states of reaction systems before and after a reaction knockout is applied. We use abstract interpretation to reason about the effects of knockouts in the absence of kinetic information. Therefore, we introduce a small abstract domain that works on relations of pairs of non negative real numbers and abstract operators that represent an over-approximation of real domain addition and multiplication. We then propose a small set of properties for kinetics functions that are general enough to be also fulfilled by mass action kinetics and Michaelis-Menten kinetics. As our central contribution, we show that the predictions obtained with our abstract interpretation are independent of any particular choice of kinetics functions, as long as these fulfill these properties. Our predictions remain candidates due to the over-approximating nature of our approach. We also present an operational semantics for our approach based on a mapping of the abstract reaction semantics to constraint satisfaction problems [33].

We start off by introducing the deterministic and steady state semantics of reaction systems (Section 2). Then we formalize the reaction knockout task in terms of a real domain constraint satisfaction problem (Section 3). To do so, in the first instance, we restrict ourselves to reactions with mass action kinetics.

Then we define our abstract domain and the abstract interpretation of our mass action knockout constraints yielding sets of finite domain constraints (Section 4). We show that our abstract interpretation is a correct over-approximation in that every solution of the real domain problem is reflected by the solutions of the abstract domain problem. We proceed by introducing properties of kinetics functions and show that these are fulfilled by mass action kinetics and Michaelis-Menten kinetics (Section 5). Then, we prove that every kinetics function with these properties yields the same results in our abstraction, which allows us to generally stick to mass action in our abstract interpretation. We next show how to solve the thus obtained table constraints (Section 6). As this may yield large sets of solutions, we propose to integrate Branch-And-Bound optimization w.r.t. different criteria. We then provide in Section 7 a real-world example based on a model from literature and our own implementation and discuss the obtained solutions. These, on one hand, confirm results already known in literature and on the other hand represent new interesting strategies that are currently evaluated in wet-lab experiments. To finish, we provide a discussion of existing approaches for prediction reaction knockouts in absence of kinetic information (Section 8). An extended version of this paper with appendices containing detailed information on proofs and on the example model is provided online.[1]

2 Reaction Networks

Let M be a finite set of chemical species (molecule types). A chemical solution S with molecules in M is a tuple $(S_m)_{m\in M}$ with $S_m \in \mathbb{R}_{\geq 0}$. We call S_m the concentration of m in S, that is the number of molecules of species m in S divided by the volume of S.

A reaction network over M is a finite set R of chemical reactions with species in M. Thereby, a chemical reaction is a rewrite rule that states how chemical solutions change over time. More formally, a chemical reaction r with species in M is a tuple of the the following form:

$$(r)\ m_1, \ldots, m_j : m_{j+1}, \ldots, m_k \xrightarrow{\kappa} m_{k+1}, \ldots, m_l$$

where $m_1, \ldots, m_l \in M$ and $\kappa : \mathbb{R}^k_{\geq 0} \to \mathbb{R}_{\geq 0}$ is a function. Species $m_{j+1}, \ldots, m_k$ are called the reactants of r, i.e. molecules of this type are consumed when r is applied. Species on the right hand side are called the products of r, which are produced at application time. Species $m_1, \ldots, m_j$ are called enzymes of r. They are neither consumed nor produced but may increase the rate of reaction r, which is defined by the kinetics function κ. We assume that species can only play a single role in every reaction, i.e. they are either enzyme, reactant, or product. However, in every role they might occur more than once. We write $enz_r(m)$, $react_r(m)$, and $prod_r(m)$ for the number of occurrences of m as an enzyme, reactant, or product of r.

[1] http://www.lifl.fr/BioComputing/extendedPapers/vmcai13.pdf

$$(\text{SPEC})\ \frac{m \in M}{\frac{dS_m}{dt}(t) = \sum_{r \in R}(prod_r(m) - react_r(m)) \cdot rate_{S,r}(t)}$$

$$(\text{RATE})\ \frac{r \in R \quad enz(r) = (m_1, \ldots, m_j) \quad react(r) = (m_{j+1}, \ldots, m_k)}{rate_{S,r}(t) = \kappa_r(S_{m_1}(t), \ldots, S_{m_k}(t))}$$

Fig. 1. Deterministic semantics of reaction network R over M

A kinetics function of arity k, $\kappa : \mathbb{R}^k_{\geq 0} \to \mathbb{R}_{\geq 0}$, defines the rate (propensity) at which k-tuples of molecules of S may react, in function of the concentrations of the reactant and enzyme types in the solution. If $(m_1, \ldots, m_k)$ are the enzyme and reactant types of a chemical reaction with kinetics function κ then the reaction rate for a chemical solution S is equal to $\kappa(S_{m_1}, \ldots, S_{m_k})$. We write κ_r for the kinetics function of $r \in R$.

Standard chemical reactions have two reactants, no enzymes, and follow mass action kinetics [18]. The kinetics of such standard reactions is then simply defined as the product of the concentration of its reactants times some rate constant (see the Appendix of the extended version of this paper for more details). However, as a first approximation this reaction model can be extended to arbitrary many enzymes and reactants. We denote this generalized version of the mass action kinetics function as $ma_c : \mathbb{R}^k_{\geq 0} \to \mathbb{R}_{\geq 0}$ for some constant $c \in \mathbb{R}_{\geq 0}$, such that $ma_c(x_1, \ldots, x_k) = c \cdot x_1 \cdot \ldots \cdot x_k$.

Other kinds of kinetics functions give better models of enzymatic reactions. These are justified by compositions of several standard reactions with mass-action kinetics. The most frequent example is Michaelis-Menten kinetics, which accounts for a single-reactant reaction that is triggered by a single enzyme. It is given by the kinetics function $mm_{c_1,c_2} : \mathbb{R}^2_{\geq 0} \to \mathbb{R}_{\geq 0}$, such that $mm_{c_1,c_2}(a, e) = c_1 \cdot a \cdot e/(c_2 + a)$, where a, e are the concentrations of the reactant and enzyme, respectively. Rates following Michaelis-Menten kinetics describe a saturation curve that steadily increases with the concentration of the reactant but approaches a limit depending on the enzyme concentration. Yet another interesting alternative are Hill kinetics.

Deterministic Semantics. The deterministic semantics of a reaction network R is a collection of functions $(S_m)_{m \in M}$ of type $S_m : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. The value of $S_m(t)$ defines the concentration of m in the solution at time point t, so that the solution at time point t is $(S_m(t))_{m \in M}$. If the initial solution at time point 0 is fixed then the solutions at all later time points $t > 0$ are determined by a collection of ODEs (see below). However, we might not know the initial solution in practice.

The deterministic semantics of a reaction network R over M is defined by applying rule (SPEC) to each species in M. The resulting ODEs compute the change of S_m at time point t by substituting reactants by products for all chemical reactions $r \in R$ according to their rate, see Fig. 1. Rule (RATE) defines the rate of a reaction r at time point t by $rate_{S,r}(t) = \kappa_r(S_{m_1}(t), \ldots, S_{m_k}(t))$.

$$(\text{SPEC}_{st})\ \frac{m \in M}{\sum_{r \in R} prod_r(m) \cdot rate_{S,r} = \sum_{r' \in R} react_{r'}(m) \cdot rate_{S,r'}}$$

$$(\text{RATE}_{st})\ \frac{r \in R \quad enz(r) = (m_1, \ldots, m_j) \quad react(r) = (m_{j+1}, \ldots, m_k)}{rate_{S,r} = \kappa_r(S_{m_1}, \ldots, S_{m_k})}$$

Fig. 2. Steady state semantics of reaction network R over M

Steady-State Semantics. The steady state semantics assumes that reaction networks reach a fixed point in which all reactions continue to perform with constant speed. This means that the changes for all $m \in M$ become zero:

$$\frac{dS_m}{dt}(t) = 0$$

The amounts $S_m(t)$ will thus become constant for all species $m \in M$, so that we can denote them by S_m. Furthermore, reactions $r \in R$ become constant fluxes with constant rates, so that $rate_{S,r}(t)$ becomes constant and can thus be denoted by $rate_{S,r}$.

The steady state semantics of a reaction network R is given by the system of arithmetic equations in Fig. 2. These equations relate molecule concentrations S_m to reaction rates $rate_{S,r}$. Rule (SPEC_{st}) states that the production and consumption must coincide for any molecule type. It should be noticed that these equations are only a necessary condition for steady states of the dynamic system (and that some systems may not have any steady state). However, as mentioned earlier, we assume that reaction networks always reach a steady state.

Example. We consider a reaction network where A's and B's are inputs from the environment that can react to a complex C which is then released into the environment. We assume that all inputs and outputs are done such that an equilibrium must be reached.

$$\begin{array}{lllllll} (r_1) & \xrightarrow{ma_1} A & (r_2) & \xrightarrow{ma_2} B & (r_3)\ A, B & \xrightarrow{ma_3} C \\ (r_4)\ A & \xrightarrow{ma_4} & (r_5)\ B & \xrightarrow{ma_5} & (r_6) \quad C & \xrightarrow{ma_6} \end{array}$$

Note that we chose artificial rate constants i for reaction r_i. In practice the situation will be even worse in that most rate constants will be unknown. The deterministic semantics is given by the following system of ODEs:

$$\begin{aligned} \tfrac{dS_A}{dt}(t) &= 1 - 3 \cdot S_A(t) \cdot S_B(t) - 4 \cdot S_A(t), \\ \tfrac{dS_B}{dt}(t) &= 2 - 3 \cdot S_A(t) \cdot S_B(t) - 5 \cdot S_B(t), \\ \tfrac{dS_C}{dt}(t) &= 3 \cdot S_A(t) \cdot S_B(t) - 6 \cdot S_C(t). \end{aligned}$$

In order to determine the functions S_A, S_B, and S_C, it is sufficient to fix the initial solution. We will choose $S_A(0) = S_B(0) = S_C(0) = 0$ for illustration,

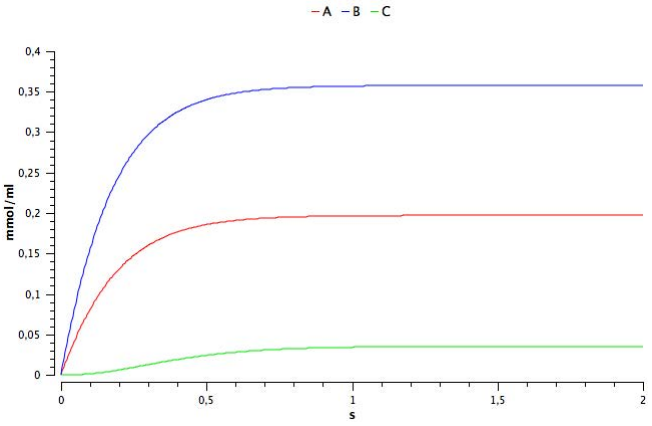


Fig. 3. Dynamics of example reaction network

which leads to the dynamics drawn in Fig. 3. It should be noticed that the concentrations of all molecules stabilize when time t tends to infinity, i.e., a steady state solution is reached. The ODEs induce the following arithmetic equations for this steady state:

$$1 = 3 \cdot S_A \cdot S_B + 4 \cdot S_A, \qquad 2 = 3 \cdot S_A \cdot S_B + 5 \cdot S_B, \qquad 3 \cdot S_A \cdot S_B = 6 \cdot S_C.$$

One can now solve these quadratic equations to determine two solutions for S_A:

$$(1)\ S_A = \frac{-23 + \sqrt{769}}{24} \qquad (2)\ S_A = \frac{-23 - \sqrt{769}}{24}$$

As only the second solution is positive, we obtain a single steady state at $S_A = 0.197119$, $S_B = 0.357695$, and $S_C = 0.0352542$.

3 Reaction Knockouts

We are now interested in modifying reaction networks, such that the rate of some reactions are increased or decreased in the steady state. The only modifications that we permit are reaction knockouts, i.e. inactivation of some reactions. As mentioned in introduction, we assume in this section that all reactions have mass action kinetics. In Section 5, we extend our approach to a more general, only partially known kinetics, relying on our abstract interpretation given in Section 4.

The knockout problem that we want to study is the following: we are given a reaction network R and some objective O. An objective compares the rate of reactions in steady states that R reaches before and after applying some reaction knockouts. O may state that the rate of some reactions r should be increased (denoted $inc(r)$), decreased ($dec(r)$), that a reaction may not be knocked out

$$(r_1)\ \xrightarrow{ma_2} A \qquad (r_2)\ A \xrightarrow{ma_1} B \qquad (r_4)\ B \xrightarrow{ma_1}$$
$$(r_3)\ A \xrightarrow{ma_1} C \qquad (r_5)\ C \xrightarrow{ma_1}$$

$$\bigwedge_{i=1}^{5} x_{on_{r_i}} \in \{0,1\} \qquad x_{r_3} = x_A \cdot x_{on_{r_3}} \qquad x_{r_1} = x_{r_2} + x_{r_3}$$
$$\exists x.(x_{r_1} = x \cdot x_{on_{r_1}} \wedge x = 2) \qquad x_{r_4} = x_B \cdot x_{on_{r_4}} \qquad x_{r_2} = x_{r_4}$$
$$x_{r_2} = x_A \cdot x_{on_{r_2}} \qquad x_{r_5} = x_C \cdot x_{on_{r_5}} \qquad x_{r_3} = x_{r_5}$$

Fig. 4. Example of a simple knockout problem: the knockout problem on top with its reaction network and objective O, its knockout constraint below

$$(\text{SPEC}_{ko})\ \frac{m \in M}{\sum_{r \in R} prod_r(m) \cdot rate^{on}_{S,r} = \sum_{r' \in R} react_{r'}(m) \cdot rate^{on}_{S,r'}}$$

$$(\text{RATE}_{ko})\ \frac{r \in R \quad \kappa_r = ma_{c_r} \quad enz(r) = (m_1, \ldots, m_j) \quad react(r) = (m_1, \ldots, m_k)}{rate^{on}_{S,r} = c_r \cdot S_{m_1} \cdot \ldots \cdot S_{m_k} \cdot on_r}$$

Fig. 5. Knockout steady states semantics of mass action reaction networks R over M

($on(r)$), or conjunctions thereof. We then try to find a set of reactions $R' \subseteq R$ to knockout, i.e. a knockout strategy, such that there exist solutions S, S' that are steady states for R and $R \setminus R'$, respectively, and satisfy O. Notice that this knockout problem could also be defined to compare several steady states that are possibly reached before and after applying some knockout. We leave such extensions to future work, see Section 9.

As a simple example consider the reaction network in Figure 4 and the objective to increase the rate of reaction r_4. Intuitively, this can be achieved by knocking out reaction r_3, such that more of the A molecules produced by reaction r_1 are transformed to B. A higher concentration of B in return leads to an increase in the speed of reaction r_4. Another idea could be to knockout reaction r_5. Then, however, the C molecules produced by reaction r_3 are no more consumed. Consequently, the concentration of C continuously increases, such that a steady state can never be reached.

In order to generally solve the above problem, we have to reason about the steady state semantics of a system of chemical reactions and all its subsets at the same time. Therefore, we first introduce the notion of reaction knockout in the semantics (knockout steady state semantics) and then we reduce the problem to reasoning with a single set of arithmetic constraints.

Knockout Steady State Semantics. We enrich our steady state semantics, such that it supports the knockout of a subset of chemical reactions. The idea is to introduce a Boolean value $on_r \in \{0,1\}$ for all reactions $r \in R$, which expresses whether or not reaction r is switched on (that is $on_r = 1$). This leads us to the knockout steady state semantics in Fig. 5.

Arithmetic Constraints. We introduce the following set of variables:

$$V = \{x_r, x_{on_r} \mid r \in R\} \cup \{x_m \mid m \in M\}$$

$$(\text{ARITHM}_{con})\ \phi_R = \bigwedge_{m \in M} \phi_m \wedge \bigwedge_{r \in R} \phi_r$$

$$(\text{SPEC}_{con})\ \frac{m \in M}{\phi_m = \left(\sum_{r \in R} prod_r(m) \cdot x_r = \sum_{r' \in R} react_{r'}(m) \cdot x_{r'}\right)}$$

$$(\text{RATE}_{con})\ \frac{r \in R \quad \kappa_r = ma_{c_r} \quad enz(r) = (m_1, \ldots, m_j) \quad react(r) = (m_1, \ldots, m_k)}{\phi_r = (x_r = c_r \cdot x_{m_1} \cdot \ldots \cdot x_{m_k} \cdot x_{on_r} \wedge x_{on_r} \in \{0, 1\})}$$

Fig. 6. Knockout constraint ϕ_R for mass action reaction networks R

Variables x_m denote the unknown concentration S_m, variables x_r the unknown reaction rate $rate^{on}_{S,r}$, and variables x_{on_r} the unknown Boolean value on_r. We then consider the following language of arithmetic constraints where $x, y, z \in V$ and $c \in \mathbb{R}_{\geq 0}$:

$$\phi \ ::= \ x + y{=}z \mid x \cdot y{=}z \mid x{=}c \mid x \in \{0, 1\} \mid \phi \wedge \phi' \mid \exists x.\phi$$

The conditions of the knockout semantics for a mass action reaction network R can now be expressed by the arithmetic constraint ϕ_R defined in Fig. 6. Notice that the constraint given there can be flattened easily, so that it belongs to the constraint language specified above. Notice further that equalities of the form $x = y$ can be expressed by $\exists z.(x{+}z = y \wedge z = 0)$. As an exemplary mapping from a reaction network to a knockout constraint, consider the one in Figure 4. One can see that the constraint has no solution when knocking out only reaction r_5, i.e. setting $x_{on_{r_5}} = 0$, since in that case we obtain the following contradiction:

$$\begin{array}{ll}
x_{r_3} = x_{r_5} = x_C \cdot x_{on_{r_5}} = 0 & \text{by } x_{on_{r_5}} = 0 \\
\Rightarrow x_{r_3} = x_A \cdot x_{on_{r_3}} = 0 = x_A & \text{by } x_{on_{r_3}} = 1 \\
\Rightarrow x_{r_2} = x_A \cdot x_{on_{r_2}} = 0 & \text{by } x_A = 0 \\
\Rightarrow x_{r_1} = 2 \cdot x_{on_1} = 2 = x_{r_2} + x_{r_3} = 0 & \text{by } x_{on_{r_1}} = 1
\end{array}$$

Let ν be a variable assignment into the domain $\mathbb{R}_{\geq 0}$. The constraint problem that we try to solve is now as follows: given a reaction network R, with its knockout constraint ϕ_R and an objective O, find variable assignments ν, ν', such that ν, ν' satisfy ϕ_R, for all reactions $r \in R$ it holds that $\nu(x_{on_r}) = 1$, and (ν, ν') satisfies objective O. The reactions r for which it holds that $\nu'(x_{on_r}) = 0$ then define our reaction knockout strategy.

4 Abstract Interpretation

The next idea is to reason about changes in concentrations in steady states when switching off reactions. This is done by interpreting arithmetic constraints abstractly into finite domain table constraints.

$+^\alpha$	$\uparrow$	$\Uparrow$	$\downarrow$	$\Downarrow$	$\sim$	$\approx$
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow,\downarrow,\sim$	$\uparrow,\downarrow,\sim$	$\uparrow$	$\uparrow$
$\Uparrow$	sy.	$\Uparrow$	$\uparrow,\downarrow,\sim$	$\uparrow,\downarrow,\sim$	$\uparrow$	$\Uparrow$
$\downarrow$	sy.	sy.	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$\Downarrow$	sy.	sy.	sy.	$\Downarrow$	$\downarrow$	$\Downarrow$
$\sim$	sy.	sy.	sy.	sy.	$\sim$	$\sim$
$\approx$	sy.	sy.	sy.	sy.	sy.	$\approx$

$\cdot^\alpha$	$\uparrow$	$\Uparrow$	$\downarrow$	$\Downarrow$	$\sim$	$\approx$
$\uparrow$	$\uparrow$	$\Uparrow$	$\uparrow,\downarrow,\sim$	$\Downarrow$	$\uparrow$	$\approx$
$\Uparrow$	sy.	$\Uparrow$	$\Uparrow$	$\approx$	$\Uparrow$	$\approx$
$\downarrow$	sy.	sy.	$\downarrow$	$\Downarrow$	$\downarrow$	$\approx$
$\Downarrow$	sy.	sy.	sy.	$\Downarrow$	$\Downarrow$	$\approx$
$\sim$	sy.	sy.	sy.	sy.	$\sim$	$\approx$
$\approx$	sy.	sy.	sy.	sy.	sy.	$\approx$

Fig. 7. Abstraction of binary addition and multiplication functions (sy. = symmetric)

Abstract Domain & Relations. We are interested in capturing the differences between pairs of nonnegative real numbers $(u, u') \in \mathbb{R}^2_{\geq 0}$. We distinguish the cases where $u > u'$, $u < u'$, and $u = u'$, and in addition those cases where u or u' are equal to 0. More formally, we define the following set of difference relations:

$$\Delta = \{\uparrow, \downarrow, \sim, \Uparrow, \Downarrow, \approx\} \subseteq \mathbb{R}^2_{\geq 0}$$

such that the following properties hold for all $u, u' \in \mathbb{R}_{\geq 0}$:

$$\begin{array}{ll} u \uparrow u' \Leftrightarrow 0 < u < u' & u \Uparrow u' \Leftrightarrow 0 = u < u' \\ u \downarrow u' \Leftrightarrow u > u' > 0 & u \Downarrow u' \Leftrightarrow u > u' = 0 \\ u \sim u' \Leftrightarrow u = u' > 0 & u \approx u' \Leftrightarrow u = u' = 0 \end{array}$$

Lemma 1. *For any $(u, u') \in \mathbb{R}^2_{\geq 0}$, there exists a unique $\delta \in \Delta$ such that $(u, u') \in \delta$.*

Given an arithmetic relation $p \subseteq \mathbb{R}^n_{\geq 0}$, we next define an abstract relation $p^\alpha \subseteq \Delta^n$, such that for all difference relations $(\delta_1, \ldots, \delta_n) \in \Delta^n$:

$$(\delta_1, \ldots, \delta_n) \in p^\alpha \Leftrightarrow \exists (u_1, \ldots, u_n) \in p \; \exists (u'_1, \ldots, u'_n) \in p. \; \textstyle\bigwedge_{i=1}^n (u_i, u'_i) \in \delta_i$$

In particular, we define abstract multiplication and addition functions $\cdot^\alpha, +^\alpha \in \Delta^3$ from binary multiplication and addition functions, seen as ternary relations. The tables of these two relations are spelled out in Fig. 7.

Abstract Constraints. Abstract constraints are finite domain table constraints, whose variables have values in Δ, subject to constraints based on the abstract relations $+^\alpha$ and $\cdot^\alpha$. We consider the following language of abstract constraints where $x, y, z \in V$ and $\Delta' \subseteq \Delta$:

$$\psi \;::=\; +^\alpha(x, y, z) \mid \cdot^\alpha(x, y, z) \mid x \in \Delta' \mid \psi \wedge \psi'$$

We first show who to compile objectives to abstract constraints:

$$\begin{array}{ll} [\![inc(r)]\!] = x_r \in \{\uparrow, \Uparrow\} & [\![on(r)]\!] = x_{on_r} \in \{\sim\} \\ [\![dec(r)]\!] = x_r \in \{\downarrow, \Downarrow\} & [\![O \wedge O']\!] = [\![O]\!] \wedge [\![O']\!] \end{array}$$

The condition that initially all reactions are on is expressed by:

$$\bigwedge_{r \in R} x_{on_r} \in \Delta \setminus \{\approx, \Uparrow\}$$

We next use abstract interpretation in order to map arithmetic constraints to abstract constraints:

$$\begin{array}{rlrl} [\![x + y = z]\!] &= +^{\alpha}(x, y, z) & [\![x \cdot y = z]\!] &= \cdot^{\alpha}(x, y, z) \\ [\![x = c]\!] &= x \in \{\sim\}, \text{with } c > 0 & [\![x \in \{0, 1\}]\!] &= x \in \Delta \setminus \{\uparrow, \downarrow\} \\ [\![x = c]\!] &= x \in \{\approx\}, \text{with } c = 0 & [\![\phi \wedge \phi']\!] &= [\![\phi]\!] \wedge [\![\phi']\!] \\ [\![\exists x.\phi]\!] &= \exists x.[\![\phi]\!] & & \end{array}$$

Consider the constraint $x = c$ which means that x is a constant that cannot be changed. Therefore, it is interpreted as $x \in \{\sim\}$, if $c > 0$ or $x \in \{\approx\}$, else. Or consider the constraint $x + y = z \wedge y = 0$. This is expressed by the corresponding abstract constraint $+^{\alpha}(x, y, z) \wedge y \in \{\approx\}$.

A pair (ν, ν') of two variable assignments into $\mathbb{R}_{\geq 0}$ induces a variable assignment μ into Δ, such that $\mu(x)$ is the difference relation between $\nu(x)$ and $\nu'(x)$ which is unique by Lemma 1. That is for all $\delta \in \Delta$ and $x \in V$:

$$\mu(x) = \delta \Leftrightarrow (\nu(x), \nu'(x)) \in \delta$$

We say that a pair (ν, ν') satisfies ψ if and only if μ is a solution of ψ.

We are now able to compile knockout constraint satisfaction problems into abstract domains. Reconsider the example in Figure 4 with the objective to increase the rate of reaction r_4. Our objective is compiled by $x_{r_4} \in \{\uparrow, \Uparrow\}$ and the condition that initially all reactions are on is expressed by $\bigwedge_{i=1}^{5} x_{on_i} \in \Delta \setminus \{\approx, \Uparrow\}$. Furthermore, the constraint $\bigwedge_{i=1}^{5} x_{on_i} \in \{0, 1\}$ is mapped to $\bigwedge_{i=1}^{5} x_{on_i} \in \Delta \setminus \{\uparrow, \downarrow\}$, such that it holds $\bigwedge_{i=1}^{5} x_{on_i} \in \{\sim, \Downarrow\}$. To complete the translation, it only remains to replace arithmetic relations by their abstract interpretation. Solving the resulting constraint, we obtain that $x_{r_1} \in \{\sim, \Downarrow\}$, such that for any solution μ also satisfying our objective it holds that $\mu(x_{r_1}) = \sim$. Thus, there exist only two solutions, either a knockout of r_3 alone ($\mu(x_{on_3}) = \Downarrow$) or of both reactions r_3 and r_5. As for the real domain constraints, a knockout of only reaction r_5 does not give any solution, since we analogously obtain contradiction $x_{r_1} = \sim = x_{r_2} + x_{r_3} = \Downarrow$.

Correctness. We now show that the abstract interpretation is correct in that every solution of the real domain problem is reflected by the solutions of the abstract domain problem.

Proposition 1 (Correctness of abstract interpretation). *Let ϕ be an arithmetic constraint and ν, ν' variable assignments into $\mathbb{R}_{\geq 0}$. It holds that if ν, ν' satisfy ϕ then (ν, ν') satisfies $[\![\phi]\!]$.*

Proof. By induction on the definition of arithmetic constraints.

$\phi = x + y = z$ Let ν and ν' be both solutions of $x + y = z$. Then we have $\nu(x) + \nu(y) = \nu(z)$ and $\nu'(x) + \nu'(y) = \nu'(z)$. It follows from the definitions of μ and $+^\alpha$ that $(\mu(x), \mu(y), \mu(z)) \in +^\alpha$, i.e., μ is a solution of the abstract constraint $+^\alpha(x, y, z)$, i.e. of $[\![\phi]\!]$

$\phi = x \cdot y = z$ analogous to $+$.

$\phi = (x = c)$ If $c > 0$ then $\nu(x) = c = \nu'(x)$ and thus $\mu(x) = \sim$, i.e., μ satisfies $[\![\phi]\!] = x$. The case of ϕ being $x = 0$ is analogous.

$\phi = x \in \{0, 1\}$ It holds that $\nu(x)$ and $\nu'(x)$ belong to $\{0, 1\}$. There are 4 possible cases, showing that the difference relation $\mu(x)$ between $\nu(x)$ and $\nu'(x)$ must be either of $\{\Uparrow, \Downarrow, \sim, \approx\}$ and thus μ satisfies $[\![\phi]\!] = x \in \{\Uparrow, \Downarrow, \sim, \approx\}$.

$\phi = \exists x.\phi'$ We obtain that $[\![\phi]\!] = \exists x.[\![\phi']\!]$. Since ν, ν' satisfy ϕ, there exist x and $u, u' \in \mathbb{R}_{\geq 0}$, such that $\nu \cup (x, u), \nu' \cup (x, u')$ satisfy ϕ'. Thus, by induction hypothesis, it holds that $(\nu \cup (x, u), \nu' \cup (x, u'))$ satisfy $[\![\phi']\!]$ and thus that (ν, ν') satisfies $[\![\phi]\!]$.

$\phi = \phi_1 \wedge \phi_2$ Since ν, ν' satisfy both ϕ_1 and ϕ_2, the induction hypothesis provides that (ν, ν') satisfies $[\![\phi_1]\!]$ and $[\![\phi_2]\!]$. Thus, (ν, ν') also satisfies $[\![\phi_1]\!] \wedge [\![\phi_2]\!]$, which equals $[\![\phi]\!]$.

Proposition 1 states that every solution of a model in the real domain is reflected by its abstract interpretation. However, the converse does not hold. Consider, e.g., the constraint $\phi = \phi_1 \wedge \phi_2$, with $\phi_1 = (x_1 = x_2)$ and $\phi_2 = (x_1 = x_2 + x_3)$. For all ν that satisfy $\phi \wedge \phi'$, we obtain that $\nu(x_3) = 0$. However, with our abstraction interpretation, μ satisfies $[\![\phi]\!] \wedge [\![\phi']\!]$, even with $\mu(x_1) = \mu(x_2) = \mu(x_3) = \uparrow$. This is a correct approximation, since there exist pairs $(\nu_1, \nu_1'), (\nu_2, \nu_2')$, such that ν_1, ν_1' satisfy ϕ_1 and ν_2, ν_2' satisfy ϕ_2, and that, although differing, correspond both to μ. For example:

$$\begin{array}{cc} \nu_1 = \{(x_1, 1), (x_2, 1)\} & \nu_2 = \{(x_1, 2), (x_2, 1), (x_3, 1)\} \\ \nu_1' = \{(x_1, 2), (x_2, 2)\} & \nu_2' = \{(x_1, 4), (x_2, 2), (x_3, 2)\} \\ \nu_1(x_i) \uparrow \nu_1'(x_i), i \in \{1, 2\} & \nu_2(x_i) \uparrow \nu_2'(x_i), i \in \{1, 2, 3\} \end{array}$$

Such a constraint results, e.g., from applying rule (SUBST_{con}) to species A and B, considering the following cyclic reaction network:

$$(r_1)\ A \xrightarrow{ma_1} B \quad (r_2)\ B \xrightarrow{ma_1} A \quad (r_3)\ B \xrightarrow{ma_1}$$

To this end, our approach, can be improved in different ways. On one hand, additional abstract relations could be defined, e.g. for commonly occurring patterns in reaction sets, like cycles of certain length. On the other hand, different methods could be applied to simplify equations. For example, one could use Gaussian elimination to symbolically solve the system of linear equations given by ϕ_1, ϕ_2 and account for the fact that $x_3 = 0$ by adding the constraint $x_3 \in \{\approx\}$. We leave such improvements as subject to future work.

5 Abstract Kinetics Functions

In the following, we extend our approach to more general kinetics. That is, we introduce properties of kinetics functions and show how kinetics functions fulfilling these properties are treated in our abstract interpretation.

The three properties of kinetics functions we consider are continuity, strict monotonicity, and conjunctiveness. Rates following strictly monotonic kinetics increase in the concentration of any reactant or enzyme. More formally, we call a kinetics function $\kappa : \mathbb{R}^n_{\geq 0} \to \mathbb{R}_{\geq 0}$ strictly monotonic if and only if for all $x_1, \ldots, x_n, x \in \mathbb{R}_{\geq 0}$ and all $i \in \{1, \ldots, n\}$ it holds:

$$x_i < x \Rightarrow \kappa(x_1, \ldots, x_n) < \kappa(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)$$

Reactions that come with conjunctive kinetics can only perform if all its reactants and enzymes are present. More precisely, a kinetics function $\kappa : \mathbb{R}^n_{\geq 0} \to \mathbb{R}_{\geq 0}$ is called conjunctive if and only of for all $i \in \{1, \ldots, n\}$ it holds:

$$\bigwedge_{i=1}^{n} x_i \neq 0 \Leftrightarrow \kappa(x_1, \ldots, x_n) \neq 0$$

In fact, the most widely used kinetics, mass action and Michaelis-Menten, are continuous, strictly monotonic, and conjunctive.

Lemma 2. *Any mass action function ma_c (with $c > 0$) is strictly monotonic, continuous, and conjunctive.*

Proof. Clear, since, by definition, we obtain $ma_c(x_1, \ldots, x_n) = c * \prod_{i=1}^{n} x_i$ for a reactions with order n. □

Lemma 3. *The Michaelis-Menten function $mm_{c,c'}$ (with $c, c' > 0$) is strictly monotonic, continuous, and conjunctive.*

Proof. By definition, we obtain $mm_{c,c'}(x_1, x_2) = c * x_1 * x_2/(c' + x_1)$. Strictly monotonic in x_2, continuous, and conjunctive clear. Strictly monotonic in x_1 becomes obvious by:

$$\frac{c * x_1 * x_2}{c' + x_1} = \frac{c * x_1 * x_2}{x_1 * (\frac{c'}{x_1} + 1)} = \frac{c * x_2}{\frac{c'}{x_1} + 1}$$

□

We obtain that in our abstract interpretation any two kinetics functions of the same arity provide the same results.

Proposition 2. *Let $\kappa_1, \kappa_2 \subseteq \mathbb{R}^n_{\geq 0} \to \mathbb{R}_{\geq 0}$ be strictly monotonic, continuous, and conjunctive kinetics functions. It holds that $\kappa_1^\alpha = \kappa_2^\alpha$.*

The proof is elaborated in the Appendix of the extended version of this paper.

Since also mass action defines a strictly monotonic, continuous, and conjunctive kinetics function, we can represent any kinetics function by mass action in our abstract interpretation. In this way, we obtain complete independence from any kinetics information, except the three rather general properties listed above.

Corollary 1. *Every strictly monotonic, continuous, and conjunctive, n-ary kinetics function can be abstracted as abstract mass action kinetics ma_c^α of reactions with order n, with any c.*

Proof. By Lemma 2 and Proposition 2. □

6 Constraint Solving

We apply the usual strategy of constraint programming to first propagate exhaustively and then to distribute. We use the common constraint propagation rules for table constraints. Let p be either of the relations $+^\alpha$ and $\cdot^\alpha$ and assume that we have a constraint $p(x_1, x_2, x_3)$. For all variables x_i we maintain a finite domain $\Delta_i \subseteq \Delta$ of possible values. We can then reduce the domain of variables x_j as follows:

$$\frac{j \in \{1,2,3\}}{x_j \in \{\delta_j \mid (\delta_1, \delta_2, \delta_3) \in p,\ \forall i \neq j.\ \delta_i \in \Delta_i\}}$$

The number of solutions would become huge if one tries to enumerate the values of all variables of the constraints $[\![\phi_R]\!] \wedge [\![O]\!]$. The usual method to deal with this problem is to impose a quality measure on solutions and to search only for high quality solutions. First of all, the fewer reactions are knocked out the better, since knockouts in the wet lab impose high costs. Second, the fewer impact the modifications have on the input-output environment of the network, the better. Which reactions are to be considered as inputs and outputs is to be specified (and is usually evident in the applications). The speed of such reactions should change only if required by the objectives, but as few as possible otherwise. This objective can be imposed by giving smaller weights to solutions that assign abstract values $\sim$ or $\approx$ to variables x_r of reactions reactions r.

The performance of a constraint solver largely benefits from such optimality considerations by the usual means of branch and bound. That is, one maintains a lower bound for the quality of the current pre-solution and only searches for solutions that are better or equally well as any solution found previously.

Further reduction of the solution set can be achieved by adding existential quantifiers to the model. Indeed, we are only interested in optimal knockout strategies, that is in the values of the variables x_{on_r} of all reactions and the values of variables x_r of input and output reactions. All other variables define internal fluxes, so that they can be considered as existentially quantified. That is, for every choice of values for these variables we will only verify whether there exists one possible choice for the values of the other variables. From this follows that the performance of our constraint solver largely depends on the number of variables that are considered as part of the optimality criterion.

7 Leucine Overproduction: A Case Study

In this section, we apply our approach to predict knockout strategies for the overproduction of Leucine in *B. subtilis*. Our model forms the current status of our work in progress to extend the efforts of modeling the metabolism of *B. subtilis* as presented in [19].

The reaction network is given in Figure 8. Molecules types are notated by ovals and reactions by boxes, respectively. A reaction's products are denoted by outgoing arrows, reactants by continuous, and enzymes by dashed lines. Red

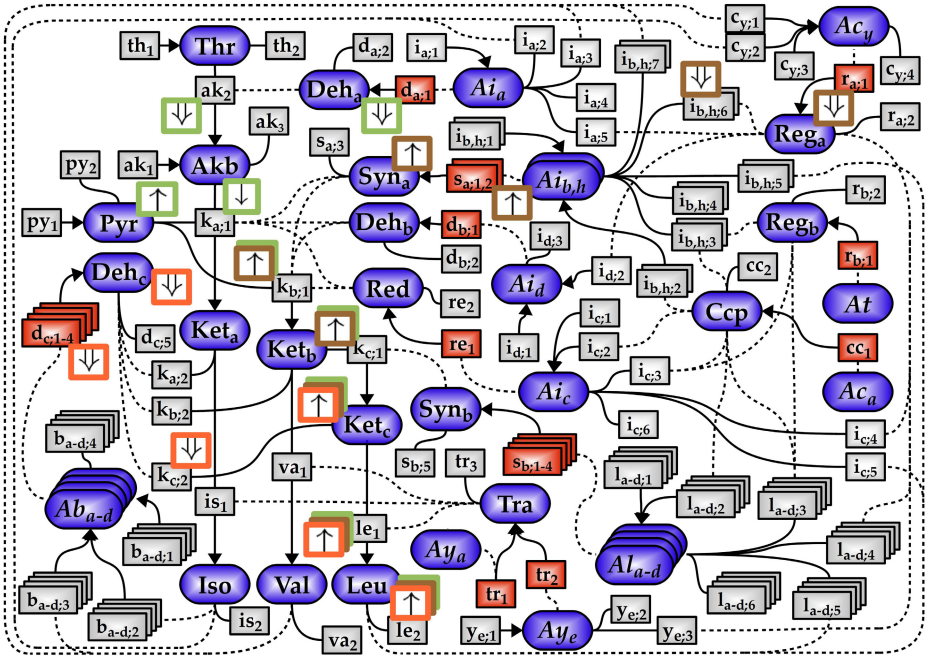


Fig. 8. A model of Leucine production in *B. subtilis* in graphical form. This model forms the current status of our work in progress to extend the efforts of modeling the metabolism of *B. subtilis* as presented in [19]. - Ovals graphically denote molecule types, rectangles reactions, arrows products and continuous and dashed lines reactants and enzymes, respectively. Red boxes mark reactions that may be knocked out. For improved readability, reactions and species are grouped where possible, which is denoted by overlapping boxes and ovals. For example the red overlapping boxes annotated by $(\mathtt{d}_{c;1-4})$ on the left hand side denote reactions $(\mathtt{d}_{c;1})\mathsf{Ab}_\mathsf{a} :\rightarrow \mathsf{Deh}_\mathsf{c}$, $(\mathtt{d}_{c;2})\mathsf{Ab}_\mathsf{b} :\rightarrow \mathsf{Deh}_\mathsf{c}$, $(\mathtt{d}_{c;3})\mathsf{Ab}_\mathsf{c} : \mathsf{Deh}_\mathsf{c}$, and $(\mathtt{d}_{c;4})\mathsf{Ab}_\mathsf{d} :\rightarrow \mathsf{Deh}_\mathsf{c}$. The objective is to increase the rate of reaction $(\mathtt{le}_2)$. Arrows in boxes of different colors represent different knockout strategies.

boxes mark reactions that may be knocked out. For improved readability, reactions and species are grouped where possible, which is denoted by overlapping boxes and ovals. For example the red overlapping boxes annotated by $(\mathtt{d}_{c;1-4})$ on the left hand side denote reactions $(\mathtt{d}_{c;1})\mathsf{Ab}_\mathsf{a} :\rightarrow \mathsf{Deh}_\mathsf{c}$, $(\mathtt{d}_{c;2})\mathsf{Ab}_\mathsf{b} :\rightarrow \mathsf{Deh}_\mathsf{c}$, $(\mathtt{d}_{c;3})\mathsf{Ab}_\mathsf{c} : \mathsf{Deh}_\mathsf{c}$, and $(\mathtt{d}_{c;4})\mathsf{Ab}_\mathsf{d} :\rightarrow \mathsf{Deh}_\mathsf{c}$. Our objective is to increase the rate of the Leucine secreting reaction $(\mathtt{le}_2)$. The reaction network in textual form and a legend for abbreviated species names is provided in the Appendix of the extended version of this paper.

In a first experiment, the goal was to find knockout strategies that are optimized w.r.t. a minimal number of knockouts. Based on our implementation, we could solve this task in about 5s on a Dell Latitude E6320 machine (Intel Core i7-2640M CPU, 2.8 GHz, 8 GB of memory).

Several single knockout solutions were proposed. Explanations for three of them are graphically annotated in Figure 8 by orange, brown, and green boxed arrows, respectively. The first one is to knock out one of the four reactions $(\mathtt{d}_{\mathtt{c};1-4})$ (orange). This leads to the removal of species $\mathsf{Deh_c}$ and therefore to a stop of reaction $(\mathtt{k}_{\mathtt{c};2})$, i.e. the degradation of species $\mathsf{Ket_c}$. In this way the rate of reaction $(\mathtt{le}_1)$ increases and so do the concentration of species Leu and the rate of reaction $(\mathtt{le}_2)$.

Knocking out reaction $(\mathtt{r}_{\mathtt{a};1})$ (brown) leads to a total lack of $\mathsf{Reg_a}$. Since this is involved in the degradation of $\mathsf{Ai_b}$ and $\mathsf{Ai_h}$ (reactions $(\mathtt{i}_{\mathtt{b};4})$ and $(\mathtt{r}_{\mathtt{a};1})$), the concentrations of $\mathsf{Ai_b}$ and $\mathsf{Ai_h}$ increase. Consequently, reactions $(\mathtt{s}_{\mathtt{a};1})$ and $(\mathtt{s}_{\mathtt{a};2})$ are accelerated, such that more $\mathsf{Syn_a}$ is produced. This leads to an increase in the speed of reaction $(\mathtt{k}_{\mathtt{b};1})$, by this raising the concentrations of $\mathsf{Ket_b}$, $\mathsf{Ket_c}$, and Leu. This knockout strategy confirms what is presented in literature (cf. [4], knockout of gene *codY*).

The deletion of reaction $(\mathtt{d}_{\mathtt{a};1})$ (green) also leads to an acceleration of reaction $(\mathtt{k}_{\mathtt{b};1})$. When disabling reaction $(\mathtt{d}_{\mathtt{a};1})$, $\mathsf{Deh_a}$ is entirely removed from the system, such that the transformation of Thr to Akb (reaction $(\mathtt{ak}_2)$) stops. Thus, the concentration of Akb is decreased, such that reaction $(\mathtt{k}_{\mathtt{a};1})$ is slowed down. Therefore, the concentration of Pyr is increased, augmenting the rate of reaction $(\mathtt{k}_{\mathtt{b};1})$.

We performed a second experiment, where we were interested in optimizing additionally subject to the number of side effects w.r.t. to species Thr,Akb,Pyr,Iso, and Val. Therefore, we set reactions $(\mathtt{th}_2)$, $(\mathtt{ak}_3)$, $(\mathtt{py}_2)$, $(\mathtt{is}_2)$, and $(\mathtt{va}_2)$ to be output reactions. For example the green strategy is likely to have a side effect on species Akb, because it decreases the rate of reaction $(\mathtt{ak}_2)$. Since in steady state, the sum of the rates of reactions $(\mathtt{ak}_1)$ and $(\mathtt{ak}_2)$ has to equal the sum of the rates of reactions $(\mathtt{ak}_3)$ and $(\mathtt{k}_{\mathtt{a};1})$, this may lead to a decrease of the speed of reaction $(\mathtt{ak}_3)$, which in return can only be achieved by lowering the concentration of Akb. Alternatively, the rate of reaction $(\mathtt{k}_{\mathtt{a};1})$ could be decreased by reducing the concentrations of Pyr or $\mathsf{Syn_a}$. However, this may require additional knockouts, e.g. in case of $\mathsf{Syn_a}$, and again lead to different side effects.

The computation of the experiment took about 163s on the same machine. We obtained that the orange strategy and the combination of the brown and the green strategy are valuable candidates.

We further tested the obtained strategies on false solutions by mapping them to finite integer domain problems. This was done based on the knockout semantics in Figure 6 and by encoding the results into additional constraints. That is, given the knockout constraint ϕ of our reaction network with variables $x_1, \ldots, x_n$ and solution μ, we formulated the finite integer domain problem $\phi \wedge \phi' \wedge \bigwedge_{i=1}^{n} \phi_{x_i}$ over variables $x_1, x_1', \ldots, x_n, x_n'$, where ϕ' is the constraint ϕ but with variables $x_1' \ldots, x_n'$ and ϕ_{x_i} is the encoding of the abstract value $\mu(x_i)$ into a constraint over the pair (x_i, x_i') according to our abstract interpretation of real value pairs. As a result, we obtained, with variable domains $\{1, \ldots, 1000\}$, solutions for all proposed strategies, providing that none of them denotes a false abstract solution.

Although this supports the validness of our proposition, in practice the test on false solutions can in general only be considered a side note. We expect the relevance of proposed modification strategies to be more significantly impacted by the validity of models which needs to be continuously improved and should eventually lead to the integration of further available biological information, such as gene expression or flux data [34] (see also Section 9). Currently, the combination of the green and the brown strategy is tested in wet-lab experiments.

8 Related Work

Most existing approaches to solve reaction knockout problems formulate two-level optimization problems [5,27,28,26,36,23]. Thereby, the first level of optimization bases on the idea of flux balance analysis [39,29,3,9]. It captures only the equations stating that for any species the sum of the rates of the producing and the consuming reactions equal (Figure 2, rule (SPEC)). The constraints on reaction rates (variables $rate_{S,r}$) as introduced by rule (RATE_{st}) are omitted. Since the resulting sets of equations are hopelessly underconstraint, determining the values of variables $rate_{S,r}$ is considered to be an optimization problem with different kinds of optimization goals, e.g. biomass production (optimal growth) [38,5] or ATP production (optimal energy) [30].

The intuition behind this first level of optimization is that, considering the background of evolution, organisms are assumed to be trimmed in a way, such that they always regulate their metabolism for optimal chances of survival. The second level of optimization is then concerned with finding the gene knockout strategy that yields the highest rate for the given target reaction, as determined by the first optimization level. Such two-level optimization problems are then solved by using e.g. integer linear programming approaches [5,23] or evolutionary algorithms [26].

Predicting knockout strategies by two-level optimization is appealing, since it projets the problem to a well-founded mathematical domain. However, we see several drawbacks of this approach. On one hand, by dropping the constraints on reaction rates the relations between concentrations and reaction rates are lost. For example, the increase in the rate of a reaction is usually caused by the increase in the concentration of some reactant. If this reactant takes also part in other reactions then it is likely to cause changes in their rates, too. When not considering rate constraints such side effects will not show. This is critical, since in particular negative feed-back loops that are a common theme in reaction networks cannot be taken into account.

On the other hand, whether the assumptions used to define optimization goals are appropriate is controversial [35,31]. A major problem commonly listed is that artificially created organisms did in fact not face evolutionary pressure, such that they may control their metabolism in unexpected ways. To this end, on one hand, a reasoning is presented in [31] that is based on the maximal and minimal bounds of reaction rates. These are then obtained by a separate optimization procedure for each reaction. On the other hand, in [35], it is proposed to use the

assumption of minimization of metabolic adjustment. This approach is similar to ours in that it compares reaction rates values before and after modification. The reaction rates before modifications are obtained by applying the optimal growth assumption. The rate values resulting from modification are then optimized to diverge as few as possible from their original value.

Finally, by using optimization approaches, it is not possible to apply a local reasoning that considers parts of metabolic networks as in Section 7, where only the production branched-chain amino acids (Leucine, Valine, Isoleucine) is modeled. The reason is that optimization subjects apply to specific parts of metabolisms that thus always need to be considered, such as the Glycolysis pathway or the TCA cycle. This requires a reasoning on rather large models. Different models that aim to capture the entire metabolism of micro-organisms have been proposed so far [20,25,16]. However, with the size of models also their uncertainty level increases which in general crucially hampers the application of formal reasoning. Additionally, this approach may favour predictions which apply to different parts of a metabolism and impact the reaction network in a more global manner with unwanted side effects [15]. Furthermore, a more local reasoning is favorable because resulting predictions are easier to explain (cf. explanations given in Figure 8 by arrows in colored boxed that represent traces of in-/decreases). Such kind of explanations are essential in order to communicate prediction results to experts from the domain of biology and to convince them of their validity.

With other methods in the field of analyzing chemical reaction networks based on abstract interpretation [10,12,6], our approach has in common the abstraction of value domains. This can be seen by lifting our abstraction to sets of solutions: reconsider the knockout constraint semantics for reaction networks as provided in Figure 6. Let $\wp(A)$ be the power set of set A and let $A \to B$ be the set of all functions from set A to set B. Given a constraint with variable set V, the set of possible solutions of the corresponding knockout problem lies in $\wp((V \to \mathbb{R}_{\geq 0})^2)$, where $\wp$ denotes the power set. Based on this idea and the abstaction of real value pairs to abstract values as given in Section 4, we can define the abstraction function $\alpha : \wp((V \to \mathbb{R}_{\geq 0})^2) \to \wp(V \to \Delta)$ and the concretization function $\gamma : \wp(V \to \Delta) \to \wp((V \to \mathbb{R}_{\geq 0})^2)$. From α, γ, and the partial order given by set inclusion $\subseteq$, we can then build the usual Galois connection [7].

However, the major difference to other approaches denotes their focus on dynamics, i.e. the consideration of changes in molecule amounts over time [10,12,6]. The idea is that vectors of species concentrations provide states and occurences of reactions represent state transitions. In [12,10], e.g., this state space is understood to provide a small step semantics which is then usually abstracted by a collecting semantics. By contrast, we only consider a single state change (between two steady states) that is directly encoded into our abstract domain and waive the idea of abstracting small step semantics.

9 Conclusion and Future Work

We have introduced an approach for predicting knockout strategies in reaction networks with partial kinetic information, based on abstract interpretation and constraint solving. We showed that our approach is independent of any particular choice of kinetics functions, as long as these are continuous, strictly monotonic and conjunctive. Our predictions remain candidates due to the over-approximating nature of our abstraction.

A major subject for future work is to find ways to reduce the number of false solutions. On one hand, we plan to integrate methods to simplify systems of equation, linear and non-linear, as they result from our knockout semantics. On the other hand, we would like to come up with less aggressive abstract interpretations, so that one can predict weights of knockout effects. To this end, sources for more detailed kinetic information shall be developed, e.g. gene expression or flux data [34].

As a further subject, we also plan to integrate new optimization criteria for solutions. For example, one could consider, instead of only one, sets of solutions of the constraint satisfaction problem that correspond to the same knockout strategy. As each solution possibly represents a pair of steady states that are reached before and after a knockout is applied, it would make sense to favor, e.g., those knockout strategies that provide an optimal ratio between the numbers of solutions that fulfill an objective and those that do not.

We hope that the provided methods will help us to obtain better knockout results for wet-lab engineering. Concrete case studies are on the way. We also hope that better prediction methods will increase the interest in improving the quality of existing reactions networks in synthetic biology.

Acknowledgments. We would like to thank Jérôme Feret and the anonymous reviewers for their valuable comments. This work was partly fundeded by the projects Iceberg ANR-IABI-3096 and BQR University of Lille 1 "Biologie Synthétique pour la Synthèse Dirigée de Peptides Microbiens Bioactifs".

References

1. Andrianantoandro, E., Basu, S., Karig, D.K., Weiss, R.: Synthetic biology: new engineering rules for an emerging discipline.. Molecular Systems Biology 2(1), msb4100073–E1–msb4100073–E14 (2006)
2. Benner, S.A., Michael Sismour, A.: Synthetic biology. Nature Reviews Genetics 6(7), 533–543 (2005)
3. Bonarius, H.P.J., Schmid, G., Tramper, J.: Flux analysis of underdetermined metabolic networks: the quest for the missing constraints. Trends in Biotechnology 15(8), 308–314 (1997)
4. Brinsmade, S.R., Kleijn, R.J., Sauer, U., Sonenshein, A.L.: Regulation of CodY Activity through Modulation of Intracellular Branched-Chain Amino Acid Pools. J. Bacteriol. 192(24), 6357–6368 (2010)

5. Burgard, A.P., Pharkya, P., Maranas, C.D.: Optknock: a bilevel programming framework for identifying gene knockout strategies for microbial strain optimization. Biotechnology and Bioengineering 84(6), 647–657 (2003)
6. Camporesi, F., Feret, J.: Formal reduction for rule-based models. In: Mislove, M., Ouaknine, J. (eds.) The 27th Conference on the Mathematical Foundations of Programming Semantics - MFPS 2011, Pittsburgh, États-Unis. Electronic Notes in Theoretical Computer Science, vol. 276, pp. 29–59. Elsevier (September 2011)
7. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
9. Covert, M.W., Schilling, C.H., Palsson, B.: Regulation of gene expression in flux balance models of metabolism. Journal of Theoretical Biology 213(1), 73–88 (2001)
10. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Abstracting the differential semantics of rule-based models: Exact and automated model reduction. In: LICS, pp. 362–381. IEEE Computer Society (2010)
11. Elowitz, M.B., Leibler, S.: A synthetic oscillatory network of transcriptional regulators. Nature 403(6767), 335–338 (2000)
12. Fages, F., Soliman, S.: Abstract interpretation and types for systems biology. Theor. Comput. Sci. 403(1), 52–70 (2008)
13. Feret, J., Henzinger, T., Koeppl, H., Petrov, T.: Lumpability abstractions of rule-based systems. In: Theoretical Computer Science (2012)
14. Ferrell, J.E.: Feedback regulation of opposing enzymes generates robust, all-or-none bistable responses. Current biology: CB, 18(6) (March 2008)
15. Florez, L., Gunka, K., Polania, R., Tholen, S., Stulke, J.: SPABBATS: A pathway-discovery method based on Boolean satisfiability that facilitates the characterization of suppressor mutants. BMC Systems Biology 5(1), 5+ (2011)
16. Förster, J., Famili, I., Fu, P., Palsson, B.Ø., Nielsen, J.: Genome-scale reconstruction of the Saccharomyces cerevisiae metabolic network. Genome Research 13(2), 244–253 (2003)
17. Gillespie, C.S.: Moment-closure approximations for mass-action models. IET Systems Biology 3(1), 52–58 (2009)
18. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry 81, 2340–2361 (1977)
19. Goelzer, A., Brikci, F.B., Verstraete, I.M., Noirot, P., Bessieres, P., Aymerich, S., Fromion, V.: Reconstruction and analysis of the genetic and metabolic regulatory networks of the central metabolism of Bacillus subtilis. BMC Systems Biology, 2(1), 20+ (2008)
20. Henry, C.S., Zinner, J.F., Cohoon, M.P., Stevens, R.L.: iBsu1103: a new genome-scale metabolic model of Bacillus subtilis based on SEED annotations. Genome Biology 10(6), R69+ (2009)
21. Keasling, J.D.: Synthetic biology for synthetic chemistry. ACS Chemical Biology 3(1), 64–76 (2008)
22. Kholodenko, B.N.: Cell-signalling dynamics in time and space. Nature Reviews Molecular Cell Biology 7, 165–176 (2006)
23. Kim, J., Reed, J.: OptORF: Optimal metabolic and regulatory perturbations for metabolic engineering of microbial strains. BMC Systems Biology 4(1), 53+ (2010)
24. Koide, T., Pang, W.L.L., Baliga, N.S.: The role of predictive modelling in rationally re-engineering biological systems. Nature Reviews. Microbiology 7(4), 297–305 (2009)

25. Oh, Y.-K., Palsson, B.O., Park, S.M., Schilling, C.H., Mahadevan, R.: Genome-scale Reconstruction of Metabolic Network in Bacillus subtilis Based on High-throughput Phenotyping and Gene Essentiality Data. Journal of Biological Chemistry 282(39), 28791–28799 (2007)
26. Patil, K.R.R., Rocha, I., Förster, J., Nielsen, J.: Evolutionary programming as a platform for in silico metabolic engineering. BMC Bioinformatics 6(1), 308+ (2005)
27. Pharkya, P., Burgard, A.P., Maranas, C.D.: OptStrain: A computational framework for redesign of microbial production systems. Genome Research 14(11), 2367–2376 (2004)
28. Pharkya, P., Maranas, C.D.: An optimization framework for identifying reaction activation/inhibition or elimination candidates for overproduction in microbial systems. Metabolic Engineering 8(1), 1–13 (2006)
29. Price, N.D., Reed, J.L., Palsson, B.Ø.: Genome-scale models of microbial cells: evaluating the consequences of constraints. Nature Reviews. Microbiology 2(11), 886–897 (2004)
30. Ramakrishna, R., Edwards, J.S., McCulloch, A., Palsson, B.O.: Flux-balance analysis of mitochondrial energy metabolism: consequences of systemic stoichiometric constraints. American Journal of Physiology. Regulatory, Integrative and Comparative Physiology 280(3), R695–R704 (2001)
31. Ranganathan, S., Suthers, P.F., Maranas, C.D.: OptForce: An Optimization Procedure for Identifying All Genetic Manipulations Leading to Targeted Overproductions. PLoS Comput. Biol. 6(4), e1000744+ (April 2010)
32. Rodrigo, G., Carrera, J., Landrain, T.E., Jaramillo, A.: Perspectives on the automatic design of regulatory systems for synthetic biology. FEBS Letters 586(15), 2037–2042 (2012)
33. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier (2006)
34. Sauer, U.: Metabolic networks in motion: 13C-based flux analysis. Molecular Systems Biology, 2(1) (November 2006)
35. Segrè, D., Vitkup, D., Church, G.M.: Analysis of optimality in natural and perturbed metabolic networks. Proceedings of the National Academy of Sciences 99(23), 15112–15117 (2002)
36. Tepper, N., Shlomi, T.: Predicting metabolic engineering knockout strategies for chemical production: accounting for competing pathways. Bioinformatics 26(4), 536–543 (2010)
37. Thomas, R.: Boolean formalization of genetic control circuits. Journal of Theoretical Biology 42(3), 563–585 (1973)
38. Varma, A., Palsson, B.O.: Metabolic Capabilities of Escherichia coli II. Optimal Growth Patterns. Journal of Theoretical Biology 165(4), 503–522 (1993)
39. Varma, A., Palsson, B.O.: Metabolic Flux Balancing: Basic Concepts, Scientific and Practical Use. Nature Biotechnology 12(10), 994–998 (1994)
40. Wolkenhauer, O., Ullah, M., Kolch, W., Cho, K.-H.H.: Modeling and simulation of intracellular dynamics: choosing an appropriate framework. IEEE Transactions on Nanobioscience 3(3), 200–207 (2004)

Reduced Product Combination of Abstract Domains for Shapes*

Antoine Toubhans[1], Bor-Yuh Evan Chang[2], and Xavier Rival[1]

[1] INRIA, ENS, CNRS, Paris, France
[2] University of Colorado, Boulder, Colorado, USA
toubhans@di.ens.fr, bec@cs.colorado.edu, rival@di.ens.fr

Abstract. Real-world data structures are often enhanced with additional pointers capturing alternative paths through a basic inductive skeleton (e.g., back pointers, head pointers). From the static analysis point of view, we must obtain several interlocking shape invariants. At the same time, it is well understood in abstract interpretation design that supporting a separation of concerns is critically important to designing powerful static analyses. Such a separation of concerns is often obtained via a reduced product on a case-by-case basis. In this paper, we lift this idea to abstract domains for shape analyses, introducing a domain combination operator for memory abstractions. As an example, we present *simultaneous separating shape graphs*, a product construction that combines instances of separation logic-based shape domains. The key enabler for this construction is a static analysis on inductive data structure definitions to derive relations between the skeleton and the alternative paths. From the engineering standpoint, this construction allows each component to reason independently about different aspects of the data structure invariant and then separately exchange information via a reduction operator. From the usability standpoint, we enable describing a data structure invariant in terms of several inductive definitions that hold simultaneously.

1 Introduction

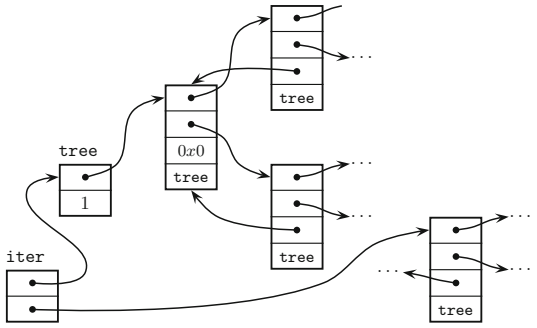
Shape analyses aim at inferring precise and sound invariants about programs manipulating complex data structures so as to prove safety and functional properties [20,10,2,5]. Such data structures typically mix several independent characteristics, which makes abstraction hard. For instance, an extreme case would be that of a binary tree with parent pointers, satisfying a balancing property and sortedness of data fields, and the property that all nodes contain a pointer to some shared record. Some shape analysis tools rely on a hard-coded abstraction [10,2], while others require some specialization of a generic abstract

* The research leading to these results has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD and the United States National Science Foundation under grant CCF-1055066.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 375–395, 2013.

```
typedef struct node {
    struct node * l;
    struct node * r;
    struct node * p;
    struct tree * h;
} * node;
typedef struct tree {
    struct node * r;
    int i;
} * tree;
typedef struct iterator {
    struct tree * t;
    struct node * c;
} * iter;
```

(a) Type definitions



(b) A concrete instance

Fig. 1. A complex tree structure

domain, either via user-supplied instrumentation predicates or inductive definitions [20,5]. In either case, when the data structures of interest feature many independent characteristics as in the aforementioned example, the user-supplied specifications or hard-coded abstractions should ideally reflect some level of *separation of concerns*, for example, between balancing properties, parent pointers, and properties over keys.

Intuitively, a separation of concerns in a shape analysis means that it should be possible for conceptually independent attributes of a data structure to be understood and treated separately by both the tool and the tool user. As an example, Fig. 1 shows an iterator over a binary tree with parent pointers. The `node` data-type (Fig. 1(a)) features left (`l`), right (`r`), and parent (`p`) pointers, as well as a special field `h` pointing to an enclosing record; the purpose of this record is to keep track of the root node of the whole structure (field `r`) and the number of active iterators (field `i`). Iterator `iter` encloses pointers to the structure being iterated over (field `t`) and to the current position (field `c`). All nodes satisfy two independent invariants: (1) `p` fields point to their parent and (2) `h` fields point to the enclosing record `tree`.

Reduced product [8] of abstract domains [7] is a framework achieving a separation of concerns in a static analyzer by combining several abstractions $\mathbb{D}_0, \ldots, \mathbb{D}_k$ and expressing conjunctions of abstract properties of the form $p_0 \wedge \ldots \wedge p_k$, where p_i is an abstract element of abstract domain $\mathbb{D}_i$. This construction has been abundantly used to combine *numerical* abstract domains into more expressive ones without having to design a monolithic domain that would be overly complex to set up and implement. For example, the Astrée analyzer [3] has been built as such a combination of numeric and symbolic abstract domains [9]; reduced product is implemented as a generic operation over abstract domains. While shape analyses often decompose abstract properties using *separating* conjunction [10,2,5] introduced in [19], few analyses explicitly introduce *non-separating*

conjunction, as this operation is viewed as more complex: for instance, updates need be analyzed for all conjunction elements. Non-separating conjunctions tend to be used in a local manner in order to capture co-existing views over small blocks [18,14]. The work presented in [15] uses global conjunctions (in addition to other refinements), yet does not turn it into a general abstract domain operation.

In this paper, we present the following contributions:

- We set up a framework for reduced product of memory abstractions in Sect. 3.
- We instantiate our framework to separating shape graphs in Sect. 4.
- We extend this instantiation to cope with user-supplied inductive definitions in Sect. 5; in that case, the reduction functions use information computed by static analysis of the inductive definitions.

Moreover, the reduced product combinator was implemented in the MemCAD analyzer (`http://www.di.ens.fr/~rival/memcad.html`), and experiments on reduction strategies are reported in Sect. 6.

2 Analysis of an Iterator over a Tree with Parent Pointers

In this section, we overview the abstraction of the structure shown in Fig. 1 using conjunctions of simpler properties and how an iteration procedure, with imperative update, can be analyzed. For simplicity in presentation, we assume variables `tree` and `iter` have global scope. Variable `tree` points to the header structure of type `tree`, while `iter` points to an iterator of type `iterator`. The tree structure can be captured by an inductive definition that can be given as a parameter to a parametric abstract domain such as that of Xisa [5,6]. For instance, the inductive definition below captures both the tree structure, the consistency of the parent pointers, and the fact that each node is separated, as materialized by the separating conjunction operator applied between fields and inductive calls:

$$\begin{aligned}\alpha \cdot \iota_{\mathrm{p}}(\beta) ::= \quad & \alpha = 0 \wedge \mathbf{emp} \\ \bigvee \; & \alpha \neq 0 \wedge \left(\begin{array}{l} \alpha \cdot \mathtt{l} \mapsto \delta_l * \alpha \cdot \mathtt{r} \mapsto \delta_r * \alpha \cdot \mathtt{p} \mapsto \beta * \alpha \cdot \mathtt{h} \mapsto \_ \\ \quad * \, \delta_l \cdot \iota_{\mathrm{p}}(\alpha) * \delta_r \cdot \iota_{\mathrm{p}}(\alpha) \end{array} \right)\end{aligned}$$

The parameter β captures the address to which the `p` field of the α parameter should point. However, field `h` is not constrained, and the property that all `h` fields should point to enclosing `tree` record is not captured by this definition. The inductive definition below captures the property that all `h` should point to some given address represented by ϵ (whereas it ignores the parent pointer constraint on `p` fields):

$$\begin{aligned}\alpha \cdot \iota_{\mathrm{h}}(\epsilon) ::= \quad & \alpha = 0 \wedge \mathbf{emp} \\ \bigvee \; & \alpha \neq 0 \wedge \left(\begin{array}{l} \alpha \cdot \mathtt{l} \mapsto \delta_l * \alpha \cdot \mathtt{r} \mapsto \delta_r * \alpha \cdot \mathtt{p} \mapsto \_ * \alpha \cdot \mathtt{h} \mapsto \epsilon \\ \quad * \, \delta_l \cdot \iota_{\mathrm{h}}(\epsilon) * \delta_r \cdot \iota_{\mathrm{h}}(\epsilon) \end{array} \right)\end{aligned}$$

We can express the fact that α represents the address of a pointer to a node of the structure of Fig. 1 by $\alpha \cdot \iota_{\mathrm{p}}(\beta) \wedge \alpha \cdot \iota_{\mathrm{h}}(\epsilon)$ where β and ϵ represent the address

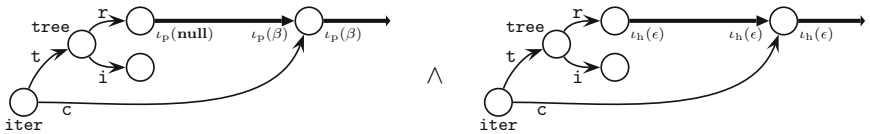
```
void next(){
  if (iter → c has a child) {
    iter → c = left child
  } else {
    go up to the first node with
      a not yet visited right child
    iter → c = that right child
  }
}

void replace(node n) {
  n → l = iter → c → l;
  n → r = iter → c → r;
  n → p = iter → c → p;
  n → h = iter → c → h;
  if (iter → c has a parent){
    update it to point n where iter → c is
  } else { iter → t → r = n; }
}

void main() {
  node n = malloc(sizeof(struct node));
  while (iter → c ≠ null && ...) { next(); }
  if (iter → t → i == 1) { replace(n); }
}
```

Fig. 2. An iteration using an iterator followed by a node replacement

of the parent pointer and of the enclosing record, respectively. We shall express such conjunctions as pairs of shape graphs, using a *product* [8] abstraction.

In the following, we consider the C program shown in Fig. 2. The code is simplified by summarizing some parts with pseudo-code. Function `main` uses the iterator to walk up to a randomly defined point of the structure, by making a series of left, right, and up steps determined by the tree structure. When no other iterator is active on `iter` → `t`, then the replacement can be performed safely, by updating fields of the node being inserted in the structure, and also fields in the parent/enclosing record depending on the position of the iterator. Note that this code reads and updates both `h` and `p` fields.

An invariant at the exit of the loop in function `main` can be represented by a conjunction of shape graphs as shown in the figure below:



The left and right shape graphs can be expressed in the abstract domain of [5], individually using only ι_p and ι_h as domain parameters, respectively. Edges abstract pairwise disjoint memory regions. Points-to edges (marked as thin edges) describe individual cells Thick edges inductively abstract (potentially empty) summarized memory regions, which could be either full structures or structure *segments*: for instance, the left shape graph expresses that field `iter`→ `c` points to a node of the tree pointed to by `tree` → `r`.

While both components of that invariant can be expressed as a shape graph in the abstract domain of [5], it is not possible to *infer* either without reasoning about parent pointers, as function `next` may follow unbounded upward paths in the tree. Similarly, the preservation of structural invariants in `replace` requires reasoning about both `p` and `h`. However, ι_p ignores information about `h` and vice versa for ι_h; thus, neither component can perform all those steps on its

own. Therefore the product analysis must organize *information exchange* among both components, which corresponds to a *reduction* operation. For instance, we consider assignment $\texttt{iter} \to \texttt{c} = \texttt{iter} \to \texttt{c} \to \texttt{p}$ in `next`. In the right component (ι_{h}), no information about `p` is known so the analysis of this operation would lose all precision when it fails to materialize this field, whereas the left component (ι_{p}) will materialize `p` with no loss in precision. The left component will also come up with the property that when $\texttt{iter} \to \texttt{c}$ has a parent, then either $\texttt{iter} \to \texttt{c} \to \texttt{p} \to \texttt{l}$ or $\texttt{iter} \to \texttt{c} \to \texttt{p} \to \texttt{r}$ is equal to $\texttt{iter} \to \texttt{c}$ (i.e., the current node is a child of its parent) that would allow a precise materialization in the right component. Similar cases occur in the analysis of `replace`. To conclude, we need to set up a language to express such simple constraints, to design operators to extract them, and to constrain abstract values with them, and to identify when such information exchange should be performed.

3 Interfaces for Memory Abstractions

In this section, we layout our framework for defining reduced products of memory abstractions. Our goal is to define a flexible abstract domain combination operator, so we begin by defining a generic interface that we expect memory abstractions to implement.

Concrete memories. We use a direct model of concrete memories m, which consist of an environment and a heap (shown inset). A *concrete environment* e is a finite map from program variables to values. A *concrete heap* h is as a finite map from addresses to values. We assume that the set of addresses is a subset of the set of values (i.e., $\mathbb{A} \subseteq \mathbb{V}$). Fields $\texttt{f}, \texttt{g}, \ldots$ are treated as numerical offsets where we write $a + \texttt{f}$ for the address that is an offset $\texttt{f}$ from base address a (i.e., $(a + \texttt{f}) \in \mathbb{A}$).

memories	$\mathbb{M} \ni m ::= (e, h)$
environments	$\mathbb{E} \ni e : \mathbb{X} \rightharpoonup_{\text{fin}} \mathbb{V}$
heaps	$\mathbb{H} \ni h : \mathbb{A} \rightharpoonup_{\text{fin}} \mathbb{V}$
variables	$x \in \mathbb{X}$
values	$v \in \mathbb{V}$
addresses	$a \in \mathbb{A}$
fields	$\texttt{f}, \texttt{g}, \ldots \in \mathbb{F}$

3.1 Memory Abstract Domains

An *abstract memory state* $m^\sharp$ describes a set of concrete memory states (shown inset). As such, it should abstract both heap addresses along with stored values. To abstract addresses and values, we let $\mathbb{V}^\sharp = \{\alpha, \beta, \ldots\}$ be a set of symbolic variables. An *abstract environment* $e^\sharp \in \mathbb{E}^\sharp = \mathbb{X} \to \mathbb{V}^\sharp$ then maps each variable into an abstraction of its address. To express the consistency

memories	$m^\sharp ::= (e^\sharp, h^\sharp)$
environments	$e^\sharp : \mathbb{X} \rightharpoonup_{\text{fin}} \mathbb{V}^\sharp$
heaps	$\mathbb{D}_\mathbf{s} \ni h^\sharp ::= \bot \mid \cdots$
concretization	$\gamma_\mathbf{s} : \mathbb{D}_\mathbf{s} \to \mathcal{P}(\mathbb{H} \times \mathcal{V})$
valuations	$\nu \in \mathcal{V}$
assignment	$\mathfrak{assign} : \mathbf{lvals} \times \mathbf{exprs} \times \mathbb{D}_\mathbf{s} \to \mathbb{D}_\mathbf{s}$
conditional	$\mathfrak{guard} : \mathbf{exprs} \times \mathbb{D}_\mathbf{s} \to \mathbb{D}_\mathbf{s}$
widening	$\triangledown : \mathbb{D}_\mathbf{s} \times \mathbb{D}_\mathbf{s} \to \mathbb{D}_\mathbf{s}$

$h^\sharp ::=$		*abstract heaps*	$\gamma_{\mathbf{s}}(h^\sharp)$
	emp	empty heap	$\{ ([], \nu) \}$
$\mid$	$\alpha \cdot \mathtt{f} \mapsto \beta$	single cell	$\{ ([\nu(\alpha) + \mathtt{f} \mapsto \nu(\beta)], \nu) \}$
$\mid$	$h_0^\sharp * h_1^\sharp$	disjoint regions	$\{ (h_0 \uplus h_1, \nu) \mid (h_0, \nu) \in \gamma_{\mathbf{s}}(h_0^\sharp) \text{ and } (h_1, \nu) \in \gamma_{\mathbf{s}}(h_1^\sharp) \}$
$\mid$	$h^\sharp \wedge P$	with constraint	$\{ (h, \nu) \mid (h, \nu) \in \gamma_{\mathbf{s}}(h^\sharp) \text{ and } \nu \models P \}$
$P ::=$		*pure predicates*	$\nu \models P$
	$\alpha = 0 \mid \alpha \neq 0 \mid P_0 \wedge P_1 \mid \ldots$		

Fig. 3. A shape abstract domain based on exact separating shape graphs

$h^\sharp ::=$	*abstract heaps*	$\gamma_{\mathbf{s}}(h^\sharp)$
$\{ \alpha \cdot \mathtt{f} \Rrightarrow \beta, \ldots \}$	set of may points-to	$\{ (h, \nu) \mid a + f \mapsto v \in h$ implies $\alpha \cdot f \Rrightarrow \beta \in h^\sharp$ and $a \in \nu(\alpha)$ and $v \in \nu(\beta) \}$

Fig. 4. A shape abstract domain based on points-to graphs

between an abstract environment and a concrete environment, we need a *valuation* ν that relates an abstract address to a concrete one. An abstract heap $h^\sharp$ expresses pointer relationships between abstract addresses, so it abstracts a set of concrete-heap–valuation pairs. A *shape abstract domain* is a set $\mathbb{D}_{\mathbf{s}}$ of abstract heaps, together with a concretization function $\gamma_{\mathbf{s}}$ and sound abstract operators. Among the abstract operators, we include operators to compute abstract post-conditions, such as $\mathfrak{assign}$ for assignments and $\mathfrak{guard}$ for conditional guards. We also include a widening operator $\triangledown$ that joins abstract states while enforcing termination of abstract iteration [7]. All of these operators are required to satisfy the usual soundness conditions—that is, they ensure that no concrete behavior will be lost in the abstract interpretation. For example, the widening operator $\triangledown$ should soundly over-approximate unions of concrete states (i.e., $\gamma_{\mathbf{s}}(h_0^\sharp) \cup \gamma_{\mathbf{s}}(h_1^\sharp) \subseteq \gamma_{\mathbf{s}}(h_0^\sharp \triangledown h_1^\sharp)$ for all abstract heaps $h_0^\sharp$ and $h_1^\sharp$). We also let $\bot$ denote the least element of the memory abstraction, which should have the empty concretization (i.e., $\gamma_{\mathbf{s}}(\bot) = \emptyset$).

Example 1 (Exact separating shape graphs). In Fig. 3, we describe exact separating shape graphs, which is a memory abstraction with no summaries. We consider separating shape graphs with inductive summaries in Sect. 5. An abstract heap $h^\sharp$ is a formula syntactically formed according to the given grammar. We define the concretization of $h^\sharp$ inductively on the formula structure in the rightmost column. Intuitively, an abstract heap is simply a finite separating conjunction [19] of must points-to predicates along with pure constraints over heap values. The formula **emp** is the abstract heap corresponding to the concrete empty heap $[]$. Thus, notice $\gamma_{\mathbf{s}}(\mathbf{emp})$ is independent of the choice of valuation ν. A must points-to $\alpha \cdot \mathtt{f} \mapsto \beta$ corresponds to a singleton heap whose address and

contents are given by the valuation ν. Here, we let valuations be functions from symbolic variables to concrete values (i.e., $\nu : \mathbb{V}^\sharp \to \mathbb{V}$). As usual, the formula $h_0^\sharp * h_1^\sharp$ joins disjoint abstract sub-heaps. In the concrete, we write $h_0 \uplus h_1$ to join sub-heaps if their *domains* are disjoint (and undefined otherwise). Finally, the formula $h^\sharp \wedge P$ conjoins a pure constraint; we write $\nu \models P$ to say valuation ν semantically entails pure predicate P.

Example 2 (May points-to graphs). As another example, we formalize a memory abstraction based on points-to graphs as one obtains from Andersen's analysis [1] in our framework (Fig. 4). An abstract heap $h^\sharp$ is a set of may points-to edges of the form $\alpha \cdot f \Rightarrow \beta$. For this abstraction, an abstract location α corresponds to a set of concrete addresses (thus, $\nu : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$). A concrete heap h is in the concretization of an abstract heap $h^\sharp$ if and only if for all concrete cells $a+f \mapsto v$ in h, there exists a corresponding may points-to edge in the abstract heap $h^\sharp$ as given by the valuation ν. In the literature, there are usually some additional constraints placed on abstract locations. These restrictions on abstract locations can be reflected as constraints on valuations ν, such as non-empty corresponding concrete addresses (i.e., $|\nu(\alpha)| \geq 1$ for all $\alpha \in \mathrm{dom}(\nu)$) and disjoint abstract locations (i.e., $\nu(\alpha) \cap \nu(\beta) = \emptyset$ for all $\alpha, \beta \in \mathrm{dom}(\nu)$). Sometimes, abstract locations are also classified as *non-summary* versus *summary*. A non-summary abstract location α means we restrict ν such that $|\nu(\alpha)| = 1$. In contrast to exact separating shape graphs, may points-to graphs can never give precise information about the *presence* of a cell. For example, observe that the empty concrete heap $[\,]$ is in the concretization of all may points-to graphs.

3.2 Products of Memory Abstractions

Shape domains implementing the interface described in Sect. 3.1 can be combined into *product abstractions* in a straightforward manner. Let us assume two shape abstract domains $\mathbb{D}_0$ and $\mathbb{D}_1$ are given. Then, $\mathbb{D}_\times \stackrel{\text{def}}{=} \mathbb{D}_0 \times \mathbb{D}_1$ has a concretization function $\gamma_\times : \mathbb{D}_\times \to \mathcal{P}(\mathbb{H}^\sharp \times \mathcal{V})$ defined by $(h^\sharp, \nu) \in \gamma_\times(h_0^\sharp, h_1^\sharp) \iff (h^\sharp, \nu) \in \gamma_0(h_0^\sharp) \wedge (h^\sharp, \nu) \in \gamma_1(h_1^\sharp)$. This amounts to expressing *non-separating conjunctions* of abstract predicates of $\mathbb{D}_0$ and $\mathbb{D}_1$.

A direct implementation of the abstract operators ($\mathfrak{assign}, \mathfrak{guard}, \triangledown, \ldots$) can be obtained by composing the underlying operators pair-wise. However, the resulting analysis will not take advantage of the information available into one abstract domain in order to refine the facts in the other domain.

To overcome that limitation, we now propose to extend the product abstract domain with a *reduction operation*. A classical (and trivial) reduction operator would map, for example, $(\bot, h_1^\sharp)$ into $(\bot, \bot)$. In this paper, we describe much more powerful reduction operators for memory abstractions that allow us to transfer non-trivial information from one shape abstract domain to another (and vice versa).

To extend domain $\mathbb{D}_0 \times \mathbb{D}_1$ into a reduced product domain $\mathbb{D}_\bowtie \stackrel{\text{def}}{=} \mathbb{D}_0 \bowtie \mathbb{D}_1$, we need to augment it with a *reduction operator* $\pi : \mathbb{D}_\bowtie \to \mathbb{D}_\bowtie$, which satisfies the following soundness condition: $\forall h_0^\sharp \in \mathbb{D}_0,\ h_1^\sharp \in \mathbb{D}_1,\ \gamma_\times(h_0^\sharp, h_1^\sharp) \subseteq \gamma_\times(\pi(h_0^\sharp, h_1^\sharp))$.

constraints	$\mathcal{F} \subseteq \mathbb{P}_f$
extract	$\mathfrak{extract} : \mathbb{D}_\mathbf{s} \to \mathcal{P}(\mathbb{P}_f)$
constrain	$\mathfrak{constrain} : \mathbb{D}_\mathbf{s} \times \mathcal{P}(\mathbb{P}_f) \to \mathbb{D}_\mathbf{s}$

To implement a reduction operator π, we need be able to extract information from one domain and forward that information as a constraint into the other. Thus, we need to set up a language of constraints $\mathbb{P}_f$ and to extend the abstract domain interface with operators $\mathfrak{extract}$ to extract constraints from an abstract value and $\mathfrak{constrain}$ to constrain an abstract value with a constraint. The language $\mathbb{P}_f$ must be the same for *every* abstract shape domains. In practice, the user has to provide an implementation of those two operators in order to be able to use our framework. Given such operators, a reduction from the left domain into the right domain, for example, is defined as follows:

$$\pi_{0\to1}(h_0^\sharp, h_1^\sharp) \stackrel{\text{def}}{=} (h_0^\sharp, \mathfrak{constrain}_1(h_1^\sharp, \mathfrak{extract}_0(h_0^\sharp))) \ .$$

We subscript the operators to make explicit the domain to which they belong. To specify the soundness requirements, we need a concretization relation for constraints. We write $h, \nu \models \mathcal{F}$ when the pair (h, ν) satisfies all the constraints $\mathcal{F}$ (i.e., we interpret a set of constraints conjunctively). We can now specify the soundness conditions for the domain operators $\mathfrak{extract}$ and $\mathfrak{constrain}$: for all $h^\sharp \in \mathbb{D}_\mathbf{s}$ and all $\mathcal{F} \in \mathcal{P}(\mathbb{P}_f)$,

$$\forall (h, \nu) \in \gamma_\mathbf{s}(h^\sharp), \quad h, \nu \models \mathfrak{extract}(h^\sharp) \tag{1}$$

$$\forall (h, \nu) \in \gamma_\mathbf{s}(h^\sharp), \quad h, \nu \models \mathcal{F} \implies (h, \nu) \in \gamma_\mathbf{s}(\mathfrak{constrain}(h^\sharp, \mathcal{F})) \tag{2}$$

Under these soundness conditions, the operator $\pi_{0\to1}$ defined above is sound (and similarly for the analogous operator $\pi_{0\leftarrow1}$). In the above, we have focused on the soundness requirements of these operators and set up a framework for discussing them. While any sequence of $\mathfrak{extract}$ and $\mathfrak{constrain}$ satisfying these properties would yield a sound result, we have not yet discussed how to do so efficiently. And in practice, these operations must be carefully crafted to transfer just the necessary information, and we must apply them parsimoniously or on-demand to avoid making the analysis overly expensive and cluttering abstract values with facts that are not actually useful for the analysis [9]. To do so requires considering specific instantiations of this framework.

4 Instantiation to Separating Shape Graph Abstractions

In this section, we consider a first instantiation of our framework for defining reduced products of memory abstract domains. We focus on a product of two instances of $\mathbb{D}_\mathbf{g}$. While this example may look overly simple at first, it illustrates a large part of the issues brought up by the analysis discussed in Sect. 2. For the moment, inductive predicates are fully unfolded and thus not present (issues specific to inductive predicates are discussed in Sect. 5). Let us consider the abstract conjunction shown in Fig. 5. While expression $\mathtt{x}\to\mathtt{l}\to\mathtt{p}\to\mathtt{h}\to\mathtt{i}$ cannot be evaluated in either component of such an abstract state, all concrete memories corresponding to their conjunction would allow that expression to evaluate, as

Fig. 5. A simple reduction example

$$
\begin{array}{lll}
p ::= & & \text{paths} \\
 & \mid \ \emptyset & \text{empty path} \\
 & \mid \ \mathtt{f} \quad (\in \mathbb{F}) & \text{single field} \\
 & \mid \ p \cdot p & \text{concatenation} \\
a ::= & & \text{formulas } (a \in \mathbb{P}_f) \\
 & \mid \ \alpha \cdot p \triangleright \beta & \text{path equality} \\
 & \mid \ \alpha \cdot p \triangleright \mathbf{null} & \text{path to null}
\end{array}
$$

(a) Syntax

$$
\begin{array}{l}
h, \nu \models \alpha \cdot \emptyset \triangleright \mathbf{null} \iff \nu(\alpha) = 0 \\
h, \nu \models \alpha \cdot \emptyset \triangleright \beta \iff \nu(\alpha) = \nu(\beta) \\
h, \nu \models \alpha \cdot \mathtt{f} \triangleright \mathbf{null} \iff h(\nu(\alpha) + \mathtt{f}) = 0 \\
h, \nu \models \alpha \cdot \mathtt{f} \triangleright \beta \iff h(\nu(\alpha) + \mathtt{f}) = \nu(\beta) \\
h, \nu \models \alpha \cdot p_0 \cdot p_1 \triangleright \bar{\beta} \ (\text{where } \bar{\beta} \in \{\mathbf{null}\} \cup \mathbb{V}^\sharp) \\
\quad \iff \exists \delta \in \mathbb{V}^\sharp, \begin{cases} h, \nu \models \alpha \cdot p_0 \triangleright \delta \\ \wedge \ h, \nu \models \delta \cdot p_1 \triangleright \bar{\beta} \end{cases}
\end{array}
$$

(b) Semantics

Fig. 6. Language of path constraints for reducing between separating shape graphs

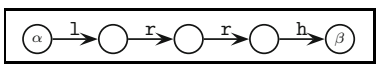
all such concrete memories would let α and δ represent the *same* address. In this section, we show how reduction allows us to perform such reasoning and to strengthen the abstract heaps of Fig. 5.

4.1 A Language of Constraints Based on Path Predicates

First of all, we notice that if (h, ν_0) belongs to the concretization of the left component, then $h(h(\nu_0(\alpha)+\mathtt{l})+\mathtt{p}) = \nu_0(\alpha)$, or equivalently, that dereferencing `l` and then `p` from address α gets us back to α. In the right component, the same sequence of dereferences yields δ: if (h, ν_1) belongs to the concretization of the right component, then $h(h(\nu_1(\alpha) + \mathtt{l}) + \mathtt{p}) = \nu_1(\delta)$. This suggests that enforcing a simple path reachability equation computed in the left component into the right component will allow us to conclude the equality of α and δ, that is, that they represent the same concrete values and can be merged.

Therefore, we include in the language of constraints to be used for reduction a way to express path predicates of the form $\alpha \cdot \mathtt{f}_0 \cdot \ldots \cdot \mathtt{f}_n \equiv \beta$ (which we write down as $\alpha \cdot \mathtt{f}_0 \cdot \ldots \cdot \mathtt{f}_n \triangleright \beta$). A *path* p is defined by a possibly empty sequence of fields, denoting field dereferences from a node, as shown in Fig. 6(a). A *path formula* a expresses that a series of dereferences described by a path will lead to read either the value denoted by some node β ($\alpha \cdot p \triangleright \beta$) or the null value ($\alpha \cdot p \triangleright \mathbf{null}$). Thus, we obtain the set $\mathbb{P}_f$ defined in Fig. 6(a). Their concretization is defined in Fig. 6(b).

As an example, path formula $\alpha \cdot \mathtt{l} \cdot \mathtt{r} \cdot \mathtt{r} \cdot \mathtt{h} \triangleright \beta$ expresses that β can be reached from α after dereferencing fields `l`, `r`, `r`, and `h` in that order, as shown in the inset figure (not all fields are represented). Intuitively



path formulas of the form $\alpha \cdot p \triangleright \beta$ allow us to express *reachability* properties, as commonly used in, for example, TVLA [20].

4.2 Reduction Operators for Shape Graphs

At this stage, we need to set up operators $\mathfrak{extract}$ and $\mathfrak{constrain}$ for the shape graph abstract domain mentioned in Sect. 3.1 using the path language of Sect. 4.1.

Triggering of the reduction process. As discussed earlier, reduction should be performed only when needed [9]. In the case of the product of shape abstract domains, information may need to be sent from one domain to another when a memory cell update or read cannot be analyzed due to a lack of pointer information:

- to read l-value $\alpha \cdot \mathtt{f}$, we need to find an edge of the form $\alpha \cdot \mathtt{f} \mapsto \beta$ in *at least one* of the components of the product; then this domain ensures the existence of the cell (even if the others fail to materialize such an edge);
- to update l-value $\alpha \cdot \mathtt{f}$, we need to find such a points-to edge in *all* components of the product; indeed, if it was not possible to do so in $\mathbb{D}_i$, then, conservative analysis of the update would require dropping *all* the information available in $\mathbb{D}_i$ since any cell may be modified from $\mathbb{D}_i$'s point of view; this would be an unacceptable loss in precision.

Therefore, a on-demand reduction strategy is to trigger when either *one* of the components fails to materialize an edge for an update or when *all* components fail to materialize an edge for a read (in the same way as for unfolding in [6]). While this is the basis of a *minimal* reduction strategy, more aggressive strategies can be used such as:

- an *on-read* strategy which attempts to reduce path constraints about any node involved in a read operation;
- a *maximal* strategy which attempts to reduce all available path constraints about all nodes, at all times.

An empirical evaluation of these strategies will be presented in Sect. 6.

Computation of path information. Operator $\mathfrak{extract} : \mathbb{D}_{\mathbf{s}} \longrightarrow \mathcal{P}(\mathbb{P}_f)$ should extract *sound* sets of constraints, satisfying soundness property (11). In the case of abstract domain $\mathbb{D}_{\mathbf{g}}$, $\mathfrak{extract}(h^\sharp)$ could simply collect all predicates of the form $\alpha \cdot \mathtt{f} \triangleright \beta$ where predicate $\alpha \cdot \mathtt{f} \mapsto \beta$ appears in $h^\sharp$:

$$\mathfrak{extract}(\alpha_0 \cdot \mathtt{f}_0 \mapsto \beta_0 * \ldots * \alpha_q \cdot \mathtt{f}_q \mapsto \beta_q) = \{\alpha_i \cdot \mathtt{f}_i \triangleright \beta_i \mid 0 \leq i \leq q\}$$

Collecting all such constraints would be almost certainly too costly, as it leads to exporting all information available in $h^\sharp$. Since our reduction reduction is triggered by a materialization operation, we only care about finding some field from some node α in either component of the product. Only constraints locally around this node are relevant, which restricts what needs to be exported. Thus, in practice, $\mathfrak{extract}$ and $\mathfrak{constrain}$ are computed locally. In the example of Fig. 5, constraint $\alpha \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \alpha$ can be extracted from the left conjunct.

$h^\sharp_\iota := h^\sharp$ (Fig. 3)	points-to, **emp**
$\mid\ h^\sharp_\iota * h^\sharp_\iota$	separating conj.
$\mid\ \alpha \cdot \iota(\beta)$	inductive predicate
$\mid\ \alpha \cdot \iota(\beta) \mathrel{-\!\!*\!\!=} \alpha' \cdot \iota(\beta')$	segment predicate

(a) Abstract heaps in $\mathbb{D}_{\langle\iota\rangle}$

- if $h^\sharp \in \mathbb{D}_{\mathbf{g}}$, $\gamma_{\langle\iota\rangle}(h^\sharp) = \gamma_{\mathbf{g}}(h^\sharp)$
- if $(h, \nu) \in \gamma_{\langle\iota\rangle}(h^\sharp_u)$ and $h^\sharp \rightsquigarrow h^\sharp_u$ then $(h, \nu) \in \gamma_{\langle\iota\rangle}(h^\sharp)$

(b) Concretization $\gamma_{\langle\iota\rangle}$

Fig. 7. A shape abstract domain based on separating shape graphs

Enforcing path information. If $\alpha \cdot p \triangleright \beta$ and $\alpha \cdot p \triangleright \gamma$, then β and γ denote the same concrete value, hence these two nodes can be merged. This rule ($\mathbf{R}_{\triangleright\equiv}$) forms the basis of an operator `constrain` satisfying soundness property (2). In the example of Fig. 5, this operator allows us to merge nodes α and δ in the right component, thanks to constraint $\alpha \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \alpha$ inferred by `extract` in the left component, as shown in the previous paragraph. This reduction allows us to materialize $\mathtt{x} \to \mathtt{l} \to \mathtt{p} \to \mathtt{h} \to \mathtt{i}$.

5 Instantiation to Separating Shape Graphs with Inductive Summaries

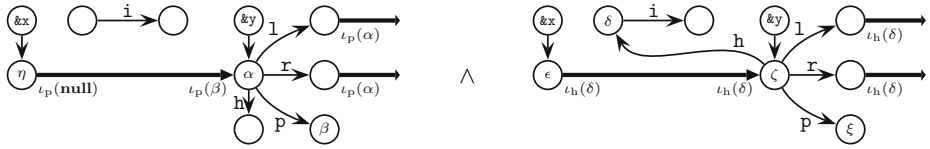
In this section, we consider a more powerful memory abstract domain, with inductive summaries for unbounded memory regions [5,6].

5.1 A Memory Abstraction with Inductive Summaries

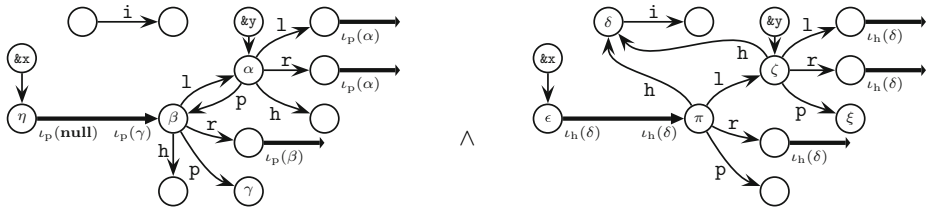
As noticed in Sect. 2, unbounded heap regions can be summarized using inductive predicates. Given an inductive definition ι, inductive predicate $\alpha \cdot \iota(\beta)$ [6], expresses that α points to a heap region containing an inductive data structure expressed by ι. Similarly, $\alpha \cdot \iota(\beta) \mathrel{-\!\!*\!\!=} \alpha' \cdot \iota(\beta')$ [6] abstracts segments of such data-structures, that is, heap regions containing a data structure with a hole. Inductive and segment predicates can be unfolded using a syntactic unfolding relation $\rightsquigarrow$, which basically unrolls inductive definitions. Therefore, assuming ι is an inductive definition, the abstract values of domain $\mathbb{D}_{\langle\iota\rangle}$ are defined as a superset of those of $\mathbb{D}_{\mathbf{g}}$, shown in Fig. 7(a). Concretization $\gamma_{\langle\iota\rangle}$ also extends $\gamma_{\mathbf{g}}$, and relies on the unfolding relation to unfold arbitrarily many times all inductive predicates into an element with no inductive predicates, which can be concretized using $\gamma_{\mathbf{g}}$.

5.2 An Extended Language of Constraints

In the following, we consider the abstract state shown in Fig. 8, which can be observed at the beginning of the `replace` function of Sect. 2. We let `y` (resp., `x`) be a shortcut for $\mathtt{iter} \to \mathtt{c}$ (resp., $\mathtt{tree} \to \mathtt{r}$). At this point, the analysis should update field $\mathtt{y} \to \mathtt{p} \to \mathtt{l}$, so this field should be materialized in all elements of the



(a) Initial abstract state



(b) Abstract state after backward unfolding

Fig. 8. Reduction example

product. Fig. 8(a) shows the abstract state before the analysis of this operation. At this stage, we notice that field $\mathtt{y}\to\mathtt{p}$ is materialized as a points-to predicate in both sides of the conjunction, but $\mathtt{y}\to\mathtt{p}\to\mathtt{l}$ is materialized in neither. To obtain this materialization, the analysis first needs to *unfold* the segment backwards (i.e., at its destination node's site) from the node corresponding to the value of y. The triggering of this unfolding in $\mathbb{D}_{\langle\iota_\mathrm{p}\rangle}$ relies on the typing of inductive parameters proposed in [6] (left conjunct in Fig. 8). That unfolding actually generates three cases: either the segment is empty, or node α is the left child of its parent, or α is the right child of its parent. As the empty segment case is trivial (it would entail that α and η are equal so that both x and y hold the same value), and the two latter cases are similar, we focus on the second one (that of a left parent). This unfolding, however, cannot be triggered in $\mathbb{D}_{\langle\iota_\mathrm{h}\rangle}$, as this domain does not capture the back-pointer tree invariant; thus the $\mathbb{D}_{\langle\iota_\mathrm{h}\rangle}$ components needs some guidance from $\mathbb{D}_{\langle\iota_\mathrm{p}\rangle}$ before it can proceed with the unfolding. Intuitively, $\mathbb{D}_{\langle\iota_\mathrm{p}\rangle}$ carries the information that any node χ in the tree that has a left child is such that $\chi\cdot\mathtt{l}\cdot\mathtt{p}\equiv\chi$. Applying this principle to node π shows that the following steps should be carried out, in the right conjunct:

- the segment of the right conjunct that ends in ζ also needs to be unfolded since $\zeta\cdot\mathtt{p}\triangleright\xi$ (as shown in the right conjunct in Fig. 8);
- after unfolding, $\pi\cdot\mathtt{l}\cdot\mathtt{p}\triangleright\xi$, thus node ξ is actually equal to node π.

This example shows that the language of constraints introduced in Sect. 4.1 needs to be extended with *universally quantified predicates* over summarized regions. As such predicates may be expressed over unbounded paths, we use general regular expressions over the set of fields (Fig. 9(a)) instead of only sequences

$$
\begin{array}{lll}
p ::= & & \text{paths} \\
\quad | & \emptyset & \text{empty path} \\
\quad | & \mathtt{f} \quad (\in \mathbb{F}) & \text{single field} \\
\quad | & p \cdot p & \text{concatenation} \\
\quad | & p + p & \text{disjunction} \\
\quad | & p^\star & \text{sequences} \\
a ::= & \ldots & \\
\quad | & a \vee a & \text{disjunction} \\
\quad | & \mathcal{S}_\forall[p, a[X]](\alpha, S) & \text{quantification}
\end{array}
$$

(a) Extended syntax

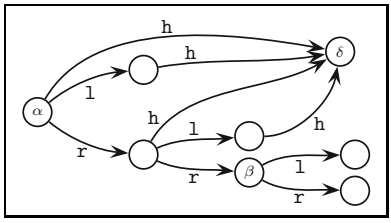
$$
\begin{array}{l}
h, \nu \models \alpha \cdot p_0 + p_1 \triangleright \beta \\
\quad \iff h, \nu \models \alpha \cdot p_0 \triangleright \beta \vee h, \nu \models \alpha \cdot p_1 \triangleright \beta \\
h, \nu \models \alpha \cdot p^\star \triangleright \beta \\
\quad \iff \exists n \in \mathbb{N},\ h, \nu \models \alpha \cdot p^n \triangleright \beta \\
h, \nu \models a_0 \vee a_1 \\
\quad \iff h, \nu \models a_0 \text{ or } h, \nu \models a_1 \\
h, \nu \models \mathcal{S}_\forall[p, a[X]](\alpha, S) \\
\quad \iff \forall \delta \in \mathbb{V}^\sharp, \begin{cases} \alpha \cdot p \triangleright \delta \Rightarrow a[\delta] \\ \vee\ \exists \beta \in S,\ \beta \cdot p \triangleright \delta \end{cases}
\end{array}
$$

(b) Extended semantics

Fig. 9. Extended language of path constraints

of offsets. Moreover, the quantification may also need be done on *segments*. Therefore, we augment $\mathbb{P}_f$ with the $\mathcal{S}_\forall[\cdot,\cdot](\cdot,\cdot)$ predicates shown in Fig. 9(a): intuitively, $\mathcal{S}_\forall[p, a[X]](\alpha, S)$ means that $a[\chi]$ holds for any node χ that can be reached from α following path expression p (i.e., $\alpha \cdot p \triangleright \delta$ holds) but cannot be reached from any node $\beta \in S$ following path expression p (i.e. $\beta \cdot p \triangleright \delta$ does not hold). The variable X is a bound variable whose scope is the disjunctive path formula a. Thus, such predicates allow us to express not only properties about inductively-summarized structures but also about segments of such structures [5,6]. The semantics of that construction is shown in Fig. 9(b), extending the definition of Fig. 6(b).

For instance, the path quantification formula $\mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{h} \triangleright \delta](\alpha, \{\beta\})$ holds true for the element of $\mathbb{D}_\mathbf{g}$ below and expresses that any node in the tree stored at address α and that is not in the sub-tree of address β for some $\beta \in S$ has an $\mathtt{h}$ field pointing to address δ, as shown in the inset figure.



5.3 Extraction of Path Predicates from Inductive Definitions

We now need to extend operators $\mathfrak{extract}$ and $\mathfrak{constrain}$ so as to allow communication from and to $\mathbb{D}_{\langle\iota\rangle}$. Compared to $\mathbb{D}_\mathbf{g}$, the main issue is that path information now has to be computed about inductive summaries; thus this computation logically requires inductive reasoning.

For instance, the property that any node α of a tree with parent pointers that has a left child is such that $\alpha \cdot \mathtt{l} \cdot \mathtt{p} \equiv \alpha$ needs be computed by induction over the ι_p inductive predicate describing that tree. This inductive reasoning can actually be done once and for all about inductive ι_p so that it can then be applied to any of its occurrences. Therefore, operator $\mathfrak{extract}$ should rely on the results of a pre-analysis of inductive definition ι_p that is computed *before* the

static analysis of the program (i.e., it may be performed only once per inductive definition used in the analyzer). Moreover, we remark that such properties can be derived by induction over the inductive definition, which suggests a fixed-point computation, that is, an abstract interpretation based static analysis of ι_{p}—a parameter to the abstract domain used to static analyze the program.

In the following, we label inductive predicates with a bound on the induction depth so as to express the soundness of the inductive definition analysis. For instance, the inductive definition ι_{p} becomes the following:

$$\begin{array}{rcl} \alpha \cdot \iota_{\mathrm{p}}^{0}(\beta) & \rightsquigarrow & \alpha = 0 \wedge \mathbf{emp} \\ \alpha \cdot \iota_{\mathrm{p}}^{i+1}(\beta) & \rightsquigarrow & \alpha = 0 \wedge \mathbf{emp} \\ & \bigvee & \alpha \neq 0 \wedge \left(\begin{array}{c} \alpha \cdot \mathtt{l} \mapsto \delta_l * \alpha \cdot \mathtt{r} \mapsto \delta_r * \alpha \cdot \mathtt{p} \mapsto \beta * \alpha \cdot \mathtt{h} \mapsto \_ \\ * \, \delta_l \cdot \iota_{\mathrm{p}}^{i}(\alpha) * \delta_r \cdot \iota_{\mathrm{p}}^{i}(\alpha) \end{array} \right) \end{array}$$

Then, the analysis should compute a sequence of sets of path predicates $(\mathcal{A}_i)_{i \in \mathbb{N}}$ such that at any rank i, if $(h, \nu) \in \gamma_{\mathbf{s}}(\alpha \cdot \iota_{\mathrm{p}}^{i}(\beta))$, then $h, \nu \models \mathcal{A}_i$.

Analysis algorithm. It proceeds by a classical abstract interpretation over the inductive structure of ι, using an abstract domain of path predicates. More precisely, given $\mathcal{A}_i$, the computation of $\mathcal{A}_{i+1}$ involves the following steps:

– *Base rules* (with no inductive call) can be handled using the $\mathfrak{extract}$ function shown in Sect. 4.2. For instance, considering ι we get $\mathcal{A}_0 = \{\alpha \cdot \emptyset \triangleright \mathbf{null}\}$.

– *Inductive rule* $\alpha \cdot \iota^{i+1}(\beta) \rightsquigarrow M * \pi_1 \cdot \iota^{i}(\rho_1) * \ldots * \pi_k \cdot \iota^{i}(\rho_k)$ (where M contains no inductive predicate) requires the following steps to be performed:

1. by instantiation of $\mathcal{A}_i$ and application of $\mathfrak{extract}$ to M, the analysis generates $\mathfrak{extract}(M) \cup \mathcal{A}^i[\pi_1, \rho_1/\pi, \rho] \cup \ldots \cup \mathcal{A}^i[\pi_r, \rho_r/\pi, \rho]$;
2. then, the analysis introduces quantified path predicates using the following principle: if $\alpha \cdot \mathtt{f}_1 \triangleright \beta_1, \ldots, \alpha \cdot \mathtt{f}_p \triangleright \beta_p$ and $a[X/\alpha]$ holds, then so does
$$\mathcal{S}_\forall[(\mathtt{f}_1 + \ldots + \mathtt{f}_p)^\star, a[X]](\alpha, \{\beta_1, \ldots, \beta_p\});$$
3. last, the analysis eliminates intermediate variables $(\pi_1, \ldots, \pi_p, \beta_1, \ldots, \beta_p)$ after applying transitivity principles to the set of path predicates:
$$\begin{array}{rl} \alpha \cdot p \triangleright \beta \wedge \beta \cdot q \triangleright \delta & \Longrightarrow \alpha \cdot p \cdot q \triangleright \delta \\ \mathcal{S}_\forall[p, a](\beta, S) \wedge \mathcal{S}_\forall[p, a](\alpha, S') & \Longrightarrow \mathcal{S}_\forall[p, a](\beta, (S \setminus \{\alpha\}) \cup S') \quad \text{if } \alpha \in S. \end{array}$$

The resulting set of path predicates $\mathcal{B}_{i+1}$ collects sound path constraints over induction depths $1, \ldots, i+1$.

– *Over-approximation* of the resulting path predicates and of those in the previous iterate $\mathcal{A}_i$, using an abstract join over sets of path predicates, defined using a rewrite relation which maps pairs of path formulas into weaker path formulas, that is, such that $\forall j \in \{0, 1\},\ a_0, a_1 \overset{\sqcup}{\mapsto} a_\sqcup \wedge h, \nu \models a_j \Longrightarrow h, \nu \models a_\sqcup$. This property is guaranteed using rules such as:

$$\begin{array}{rll} \alpha \cdot \mathtt{p} \triangleright \beta, \alpha \cdot \mathtt{q} \triangleright \beta & \overset{\sqcup}{\mapsto} \alpha \cdot \mathtt{p} + \mathtt{q} \triangleright \beta & \\ \mathcal{S}_\forall[p, a](\alpha, S), \alpha \cdot \emptyset \triangleright \beta & \overset{\sqcup}{\mapsto} \mathcal{S}_\forall[p, a](\alpha, S) & \text{if } \beta \in S \\ \mathcal{S}_\forall[p, a](\alpha, S), \mathcal{S}_\forall[p, a'](\alpha, S') & \overset{\sqcup}{\mapsto} \mathcal{S}_\forall[p, b](\alpha, S \cup S') & \text{if } a, a' \overset{\sqcup}{\mapsto} b \end{array}$$

Then, $\mathcal{A}_{i+1}$ is defined as $\{a \mid \exists (a_i, a_{i+1}) \in \mathcal{A}_i \times \mathcal{B}_{i+1},\ a_i, a_{i+1} \overset{\sqcup}{\mapsto} a\}$.

Furthermore, at each step, regular expressions may be simplified. Termination is ensured by the definition of a $\overset{\sqcup}{\mapsto}$ operator that avoids generating too large formulas and hence is a widening. The inductive definitions analysis returns a sound result, as all steps are sound, that is, they preserve the aforementioned invariant property:

Theorem 1 (soundness). *This analysis algorithm is sound:*
- *For all $i \in \mathbb{N}$ and for all $(h, \nu) \in \gamma_{\mathbf{s}}(\alpha \cdot \iota^i(\beta))$, $h, \nu \models \mathcal{A}_i$ holds.*
- *Thus, after convergence to $\mathcal{A}_\infty$, we have: $\forall (h, \nu) \in \gamma_{\mathbf{s}}(\alpha \cdot \iota(\beta)),\ h, \nu \models \mathcal{A}_\infty$.*

Path predicates derived from segments predicates. Segment predicates may be considered inductive predicates with a slightly different set of base rules for the end-point [6]. Therefore, the same inductive analysis applies to segments as well.

In particular, if we consider segment $\pi \cdot \iota_{\mathrm{p}}(\rho) \mathrel{*\!\!=\!\!\!\!-} \eta \cdot \iota_{\mathrm{p}}(\varepsilon)$, the analysis computes the iterates below:

iteration i	iterate $\mathcal{A}_i$
0	$\{\pi \cdot \emptyset \triangleright \eta, \rho \cdot \emptyset \triangleright \varepsilon\}$
1	$\left\{\begin{array}{c} \pi \cdot (\mathtt{l} + \mathtt{r})^\star \triangleright \eta\ ,\ \varepsilon \cdot \mathtt{p}^\star \triangleright \rho \\ \pi \cdot \emptyset \triangleright \eta\ \vee\ \pi \cdot \mathtt{p} \triangleright \rho\ ,\ \varepsilon \cdot \emptyset \triangleright \rho\ \vee\ \varepsilon \cdot (\mathtt{l} + \mathtt{r}) \triangleright \eta \\ \mathcal{S}_\forall[(\mathtt{l} + \mathtt{r})^\star, X \cdot \mathtt{l} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{l} \triangleright 0 \vee X \cdot \mathtt{l} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[(\mathtt{l} + \mathtt{r})^\star, X \cdot \mathtt{r} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{r} \triangleright 0 \vee X \cdot \mathtt{r} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[\mathtt{p}^\star, X \cdot \mathtt{p} \cdot (\mathtt{l} + \mathtt{r}) \triangleright X \vee X \cdot \mathtt{p} \triangleright \rho](\varepsilon, \{\rho\}) \\ \mathcal{S}_\forall[\mathtt{p}^\star, X \cdot (\mathtt{l} + \mathtt{r}) \triangleright \eta](\varepsilon, \{\rho\})\quad \ldots \end{array}\right\}$
2	$\left\{\begin{array}{c} \pi \cdot (\mathtt{l} + \mathtt{r})^\star \triangleright \eta\ ,\ \varepsilon \cdot \mathtt{p}^\star \triangleright \rho \\ \pi \cdot \emptyset \triangleright \eta\ \vee\ \pi \cdot \mathtt{p} \triangleright \rho\ ,\ \varepsilon \cdot \emptyset \triangleright \rho\ \vee\ \varepsilon \cdot (\mathtt{l} + \mathtt{r}) \triangleright \eta \\ \mathcal{S}_\forall[(\mathtt{l} + \mathtt{r})^\star, X \cdot \mathtt{l} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{l} \triangleright 0 \vee X \cdot \mathtt{l} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[(\mathtt{l} + \mathtt{r})^\star, X \cdot \mathtt{r} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{r} \triangleright 0 \vee X \cdot \mathtt{r} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[\mathtt{p}^\star, X \cdot \mathtt{p} \cdot (\mathtt{l} + \mathtt{r}) \triangleright X \vee X \cdot \mathtt{p} \triangleright \rho](\varepsilon, \{\rho\}) \end{array}\right\}$
3	$\mathcal{A}_2$ fixpoint reached

5.4 Reduction Operators in the Presence of Inductive Predicates

First, we note that the triggering and computation of reduction based on rule ($\mathbf{R}_{\triangleright\equiv}$) given in Sect. 4.2 still apply, with one minor caveat: when $\alpha \cdot p \triangleright \beta$ and $\alpha \cdot p \triangleright \gamma$, then β and γ can be merged only if path p is rigid, that is, involves no disjunction or $.^\star$ unbounded sequence. A supplementary rule ($\mathbf{R}_{\mathcal{S}\forall}$) is needed to treat universally quantified path predicates.

Triggering of the reduction process. The triggering condition for such a reduction is actually similar to before: when a field fails to be materialized at node α in the right conjunct, the reduced product searches for universal properties of the nodes of the structure to which α belongs. Thus it searches for universally quantified path properties from nodes corresponding to "ancestors" of α. In the example of Fig. 8, we notice that we need information about node ξ, which is part of the

structure pointed to by ϵ; as ϵ and η both correspond to the value stored in $\mathtt{x}$, thus denote equal values, information should be read in $\mathbb{D}_{\langle \iota_{\mathrm{p}} \rangle}$ from η.

Computation of universally quantified path information. Operator $\mathfrak{extract}$ should simply instantiate the results of the pre-analysis of inductive definitions shown in Section 5.3. In the case of the example shown in Fig. 8, node η satisfies:

$$\mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{l} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{l} \triangleright \mathbf{null}](\eta, \{\mathbf{null}\})$$

Enforcing universally quantified path information. Reduction rule $(\mathbf{R}_{\mathcal{S}\forall})$ boils down to the following principle: if $\alpha \cdot p \triangleright \delta$ and $\mathcal{S}_\forall[p, a[X]](\alpha, S)$, then either $a[\delta]$ holds or there exists $\beta \in S$ such that $\beta \cdot p \triangleright \delta$. Besides, when $S = \{\mathbf{null}\}$, we directly derive $a[\delta]$. In the right conjunct of the example of Fig. 8, as π is reachable from ϵ following a path of the form $(\mathtt{l}+\mathtt{r})^\star$, we derive $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \pi \vee \pi \cdot \mathtt{l} \triangleright \mathbf{null}$ from the above $\mathcal{S}_\forall[.,.](.,.)$ constraint. Clearly the left child of π is not null, so $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \pi$. However, $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \xi$ also holds. Thus, $(\mathbf{R}_{\triangleright\equiv})$ now applies and $\xi = \pi$. This allows us to materialize field $\mathtt{y} \to \mathtt{p} \to \mathtt{l}$ in $\mathbb{D}_{\langle \iota_{\mathrm{h}} \rangle}$. as well as in $\mathbb{D}_{\langle \iota_{\mathrm{h}} \rangle}$. The reduction allows the analysis to continue while retaining a high level of precision (note that we needed to materialize that field in both conjuncts to analyze the update precisely). Note that cases where node α would be the left child in one conjunct and the right one in the other conjunct are ruled out. Indeed, as soon as the parent nodes are identified as equal during the reduction process, the $\mathfrak{constrain}$ operator will deduce that its left and right children are equal, which would violate the separation property. A reduction in the opposite direction would need be performed for the analysis of an assignment to $\mathtt{y} \to \mathtt{h} \to \mathtt{i}$. More generally, sequences of accesses to $\mathtt{p}$ and $\mathtt{h}$ fields will generate cascades of reductions. This example shows that the reduced product analysis decomposes reasoning over $\mathtt{p}$ and $\mathtt{h}$ fields in both domains and organizes the exchange of information to help materialize points-to predicates across the product.

6 Implementation

We have implemented the memory reduced product combinator into the MemCAD analyzer (*Memory Compositional Abstract Domain*) as an ML functor taking two memory abstract domains as arguments and returning a new one and the inductive definition pre-analysis described in Sect. 5.3. Reduction strategy can be selected among those presented in Sect. 4.2 (*on-read* comes as default whereas *minimal* and *maximal* can be activated as options). The analysis is fully automatic and takes as input C code and inductive definitions such as those shown in Sect. 2. It computes a finite disjunction of abstract states for each program point. It was run on a set of over 30 micro-benchmarks, as well as medium-sized ones such as the iterator described in Sect. 2. Fig 10 presents selected analysis results (timings were measured on a 2.2 Ghz Intel Core i7 with 8 GB of RAM) that highlight the impact of the reduced product and the reduction strategies. In particular, multiple other list and trees algorithms gave similar

Filename & Description	LOCs	Reduction mode	Time (s)	(a)	(b)	(c)	(d)	Avg. disjs	Speed down
structure: *doubly linked list with shared record*									
`insert_list.c`	35	no r.p.	0.018	-	-	-	-	1.33	1
		on read	0.042	2	40	1	1	2.07	2.33
		maximal	0.056	26	417	8	4	1.96	3.11
structure: *tree with parent pointers and pointers to static record*									
`read_tree.c`	42	no r.p.	0.028	-	-	-	-	1.43	1
(random traversal		minimal	0.120	4	118	0	0	3.07	4.28
then read data field)		on read	0.086	9	391	8	0	1.87	3.07
		maximal	0.095	32	919	18	4	1.73	3.39
`insert_tree.c`	47	no r.p.	0.031	-	-	-	-	1.56	1
(random traversal		minimal	Fails						
then insert element)		on read	0.080	9	379	8	0	1.86	2.58
		maximal	0.090	43	1089	18	4	1.73	2.90
`rotate_tree.c`	47	no r.p.	0.031	-	-	-	-	1.56	1
(random traversal		on read	0.086	8	350	8	0	2.03	2.77
then rotate)		maximal	0.098	44	1201	22	4	1.92	3.16
structure: *tree with parent pointers and pointers to static record, and iterator*									
`iter_00.c`	171	no r.p.	0.278	-	-	-	-	8.74	1
(random traversal)		minimal	Fails						
		on read	0.701	64	2578	30	28	10.22	2.52
		on r & u	0.689	66	2635	28	30	9.68	2.47
		maximal	1.807	854	19714	28	30	10.06	6.5
`iter_01.c`	181	no r.p.	0.353	-	-	-	-	7.27	1
(random traversal)		on read	0.907	70	2902	34	32	7.99	2.56
		on r & u	0.871	80	3287	46	34	7.53	2.46
		maximal	2.263	978	24865	41	34	7.81	6.41

Fig. 10. Implementation Results

results and are not presented here. For each analysis, the table shows the number of LOCs, the mode (analysis with no reduced product —no r.p.—, using a monolithic domain, with a single inductive definition, or with a reduced product and minimal, on-read or maximal reduction mode), analysis time in seconds, number of calls to reduction operations in col. **(a)**, number of path predicates computed by reduction rules ($\mathbf{R}_{\triangleright\equiv}$) and ($\mathbf{R}_{\mathcal{S}\forall}$) in col. **(b)** (comprising all steps to perform reduction operations), number of node merges performed as part of reductions in col. **(c)**, number of reduction proving a disjunct has an empty concretization (hence, can be pruned) in col. **(d)**, average number of disjuncts per program point, and timing ratio compared with the analysis time with no reduction product. Reduction may prove abstract elements have an empty concretization, for example, when it infers that both $\alpha \cdot \mathtt{f} \triangleright \beta$ and $\alpha \cdot \emptyset \triangleright \mathbf{null}$ hold, or when it discovers equalities that would violate separation.

The comparison with monolithic analyses involving a single, more complex inductive definition shows those are faster than analyses with a product domain, which is not surprising, as the product analyses induces an overhead of duplicate domain operations and reduction operations. However, as regards the analysis with the "on-read" and "on r & u" strategies, the timing difference does not seem so dramatic (between 2.33X and 3.07X) and interestingly tends to reduce

on larger examples). Besides, applying the product analysis *only* to the part of the heap where the composite structure lies, using a *separating product* domain combinator would cut that cost down further (though, is not part of the scope of this work).

The key part of our empirical evaluation is to assess strategies. While an overly aggressive strategy is likely to slow down the analysis by performing useless reductions, a too passive strategy may cause a loss in precision. This is why, in some cases, the minimal reduction strategy does not allow the analysis to succeed, as it tends to perform reduction too late, at a point where a stronger `constrain` operator would be needed to fully exploit predicates brought up by `extract` (this occurs for `insert_tree.c`, `rotate_tree.c` and both analyses with the iterator structure). More eager strategies such as on-read and maximal do not suffer from this issue. The on-read strategy analyzes all tests precisely. Moreover, while the slow down of maximal over on-read is low for small programs, it tends to increase more than linearly in the analysis time and reach 6.5X on the larger examples (while the on-read strategy is around 2.5X slower), which suggests it is not likely to scale. This suggests the on-read strategy is a good balance.

Curiously, we also noticed that more aggressive strategies reduce the number of average disjuncts. Upon review, we discovered that this is due to some disjuncts being pruned by reduction earlier in the analysis and often right after unfolding points. Following this practical observation, we extended the on-read strategy so as to also perform reduction right after unfolding. The results obtained with this new strategy, called "on r & u" in the table, validate this hypothesis, as it reduces numbers of disjuncts and analysis time compared to the on-read strategy. It overall appears to be an efficient strategy.

7 Related Works

Reduced product construction has been widely studied as a mathematical lattice operator [8,11] as well as a way to combine abstract domains so as to improve the precision of static analyses [8,3,9,13,4]. However, it is notoriously hard to design a general notion of reduced product: first, optimal reduction is either not computable or too costly to compute in general (so that all reduced product implementations should instead try to achieve a compromise between the cost of reduction and precision); second, exchanging information between domains requires them to support reduction primitives using a common language that may be hard to choose depending on the application domain. We believe this is the reason why no general form of such construction has been set up for memory abstractions thus far.

The most closely related work to ours is that of [15]. In that work, the authors consider a hybrid data-structure which contains a list structure laid over a tree structure. To abstract such memory states, the authors use non-separating conjunctions over *zones*. Our construction presents some similarities to theirs: we also use non-separating conjunction. Their technique and ours are quite

complementary. Whereas our product construction focuses on the situations where precise path information must be transferred between components, theirs looks at the case where the only needed information is that two structures share the same nodes. In terms of an implementation strategy, their reduction relies on an instrumentation of the program to analyze, using ghost statements, whereas our implementation uses a *semantic* triggering of reduction, when one component fails.

Some analyses inferring properties of memory states utilize non-separating conjunctions in a local manner [18,14] in order to account for co-existing views on contiguous blocks corresponding to values of union types, or in order to handle casts of pointers to structures. Those analyses exploit conjunctions in a very local way and are unable to propagate global shape properties across the conjunctions.

Other authors have proposed to do a product of shape abstraction with a numerical domain [6,12,17]. These works are very different in that they do not combine two views of memory properties. Instead, they usually use numerical abstract values to characterize the contents of memory cells the structure of which is accounted for in the shape abstraction: thus, those are usually *asymmetric* constructions, using a form of a co-fibered domain [21], that is, where the shape abstraction "controls" the memory abstraction. Other works have examined decomposing analyses of programs with numerical and memory properties into a sequence of analyses [16]. Compared to the above works, this approach does not allow information flow between analyses into both directions.

Last, we remark that the language of constraints used for the reduction conveys reachability information, which are at the foundation of TVLA shape analyses [20]. We found it interesting to note that such predicates are effective at providing a low level view of memory properties, that is, a kind of assembly language used between shape abstractions.

8 Conclusion

In this paper, we have proposed a reduced product combinator for memory abstract domains, with a general interface, allowing a modular abstract domain design. We have shown that this product can be used with existing shape abstractions based on separation logic and inductive definitions. Moreover, we have implemented the resulting framework inside the MemCAD analyzer and shown the impact of reduction strategies on analysis results and efficiency.

A first direction for future work consists of integrating other memory abstractions into our framework, so as to benefit from the reduced product combinator with expressive abstractions, such as, a domain based on three-valued logic [20]. A second direction for future work is to design and implement a combinator for memory abstraction based on *separating* conjunction, which would enable one to apply entirely different abstractions to cope with data structures stored in different memory regions, while keeping the interactions between those abstractions minimal. Together with our reduced product combinator, this combinator would

enable one to derive analyses such as that of [15] as instances of our framework, among others, while retaining the advantages of a modular abstract domain.

References

1. Andersen, L.: Program Analysis and Specialization for the C Programming Language. PhD thesis (1994)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. CAV, pp. 178–192. Springer, Heidelberg (2007)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI, pp. 196–207 (2003)
4. Chang, B.-Y.E., Leino, K.R.M.: Abstract Interpretation with Alien Expressions and Heap Structures. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 147–163. Springer, Heidelberg (2005)
5. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape Analysis with Structural Invariant Checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
6. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: POPL, pp. 247–260 (2008)
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 (1977)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of Abstractions in the ASTRÉE Static Analyzer. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008)
10. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
11. Giacobazzi, R., Mastroeni, I.: Domain Compression for Complete Abstractions. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 146–160. Springer, Heidelberg (2002)
12. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL, pp. 239–251 (2009)
13. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: PLDI, pp. 376–386 (2006)
14. Laviron, V., Chang, B.-Y.E., Rival, X.: Separating Shape Graphs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 387–406. Springer, Heidelberg (2010)
15. Lee, O., Yang, H., Petersen, R.: Program Analysis for Overlaid Data Structures. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 592–608. Springer, Heidelberg (2011)
16. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic Strengthening for Shape Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)

17. McCloskey, B., Reps, T., Sagiv, M.: Statically Inferring Complex Heap, Array, and Numeric Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010)
18. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES, pp. 54–63 (2006)
19. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)
20. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118 (1999)
21. Venet, A.: Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 366–382. Springer, Heidelberg (1996)

Abstraction of Syntax

Vijay D'Silva* and Daniel Kroening

Department of Computer Science,
Oxford University
firstname.surname@comlab.ox.ac.uk

Abstract. The theory of abstract interpretation is a conceptual framework for reasoning about approximation of semantics. We ask if the creative process of designing an approximation can be studied mathematically. Semantic approximations, whether studied in a purely mathematical setting, or implemented in a static analyser, must have a representation. We apply abstract interpretation to syntactic representations and study abstraction of syntax. We show that semantic abstractions and syntactic abstractions are different, and identify criteria for deriving semantic abstractions by purely syntactic means. As a case study, we show that descriptions of numeric abstract domains can be derived by abstraction of syntax.

1 Where Do Abstractions Come from?

An important aspect of reasoning about the behaviour of dynamic systems such as programs, transition systems or process calculi is to focus on a few properties of interest and ignore irrelevant details. Abstract interpretation crystallises this intuition into a mathematical framework [4,5]. The first step is to characterise the behaviour of a system by fixed point in a lattice of semantic objects. The second step is to approximate this fixed point using a lattice and transformers, together called an abstract domain. The third step is to compute this fixed point approximation using iteration algorithms and operators to ensure termination and improve precision. The literature contains numerous domain-agnostic techniques for designing and computing fixed point approximations. The process of designing of an abstract domain has not been formalised in mathematical terms.

One component of designing an abstract interpreter is designing an abstract domain. The designer of an abstract domain usually has to answer three questions. What are the elements of the domain? How are abstract transformers implemented? Does analysis with the domain produce information for solving the problem? The first is a specification question, the second, an algorithmic question, and the third, an empirical evaluation question. Algorithmic details depend on the properties in the lattice, and the evaluation depends on the programs considered, so we expect that finding a generic, formal framework for answering the second and third questions will be difficult.

* Supported by Microsoft Research's European PhD Scholarship Programme.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 396–413, 2013.

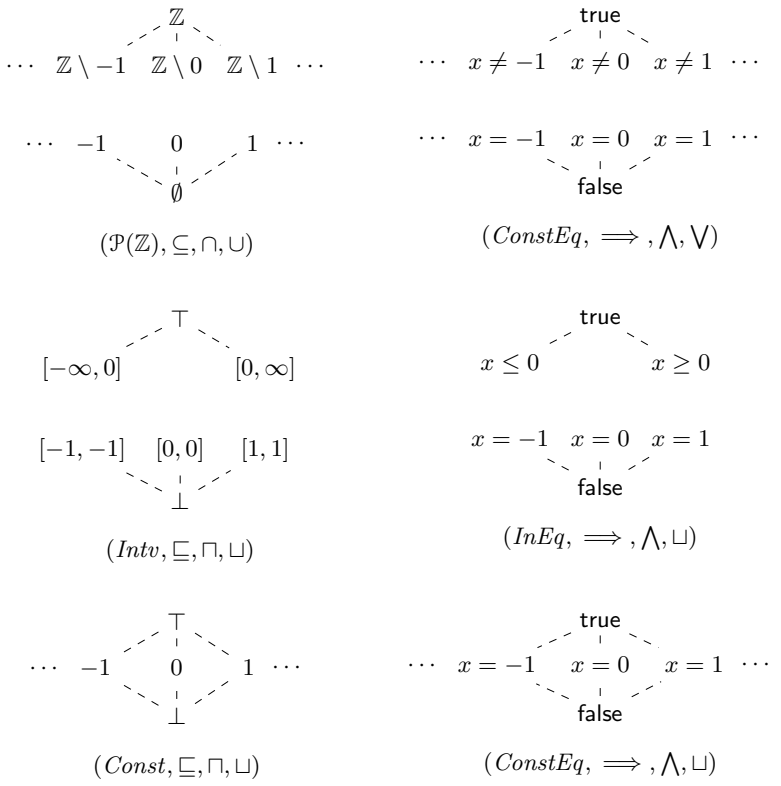


Fig. 1. The left column shows the powerset lattice of integers $\mathcal{P}(\mathbb{Z})$, the lattice of integer intervals *Intv*, and the lattice of integer constants *Const*. The right column contains logical representations of these domains. Each domain is a set of formulae ordered by implication. All sets of formulae are closed under infinitary conjunction, but not all are closed under disjunction.

We consider the problem of discovering *syntactic* specifications of abstract domains. We show that thinking of domains as logics enables a uniform approach to deriving specifications of several abstract domains used in practice. Our work applies to domains for numeric data [20], functional programs [16], and temporal logics [19]. We do not claim to formalise the process by which static analysis experts design abstractions. Instead, our work presents an alternative approach to deriving abstractions that are typically discovered manually. We believe this alternative can eventually be automated.

For an illustration of the ideas in this paper, imagine a program with a single variable x that ranges over the mathematical integers $\mathbb{Z}$. The concrete domain containing possible values of x is the lattice $\mathcal{P}(\mathbb{Z})$ shown in Figure 1. Operating on this lattice is intractable, so analysis tools use abstract domains such intervals. In the figure, each element on the left (a semantic object) is represented by a

formula on the right (a syntactic object). For instance, $x \leq 1$ represents the interval $[-\infty, 1]$. The lattice of intervals can be viewed as a logic containing formulae of the form $x \leq k$ and $x \geq k$, and closed under conjunction but not under disjunction. Implication defines the lattice order. The logics in the right-hand column of Figure 1 can be viewed as specifications of abstract domains. We study the problem of systematically discovering logical descriptions of abstract domains given a logical description of a concrete domain.

There is a loose connection between the approach we take and proof theory. Proof theory provides a mathematical basis for studying the structure and properties of formal proofs, and procedures for deriving proofs. Though proof theory does not formalise, or even automate, the exact process by which mathematicians construct proofs, it has led to new insights about the structure of proofs, and to automated deduction techniques. Analogously, this paper is a first step towards a mathematical description of how abstract domains can be derived.

One may ask if the syntactic, logical approach we take has advantages over the semantic approach usually adopted in the literature. In the Galois insertion setting, a result of Cousot and Cousot [5] shows that the space of domains is a complete lattice. Domains in this lattice can be combined using reduced sum, reduced product, reduced power, and various other operations on domains [5,12,13]. This lattice of domains is a rich mathematical object that cannot be manually searched.

In contrast, the syntax of a logic representing an abstract domain can usually be described by a short BNF grammar. Committing to a syntactic representation may restrict the kinds of domains that can be expressed, but also makes it easier to enumerate syntactic descriptions of domains. There are many parameters such as the number of operators, variables and nesting depth, that can be manipulated to generate logics from a grammar. We show later that a wide range domain specifications can be derived with our approach, and that the syntactic descriptions are succinct.

Contribution. This paper addresses the problem of deriving syntactic specifications of abstract domains. We present a rigorous, grammar-based solution to this problem. The main idea is to exploit a non-trivial connection between abstract domains and logical theories and reduce the problem of specifying domains to that of underapproximating a grammar. We make the following contributions.

- A characterisation of BNF grammars in terms of a domain of syntax trees and syntax transformers. By underapproximating the fixed point defined by a grammar, we systematically derive sub-languages.
- The derivation of overapproximating semantic domains by underapproximating the semantics of grammars.
- A case study showing that several existing domains can be derived using the framework developed in this paper.

By decoupling the problem of domain specification from that of algorithm design and evaluation, we believe intellectual effort can be directed towards the parts of the problem that are difficult to systematise. To revisit the analogy to proof

theory, we believe that a syntactic approach to abstract domain design, may lead to new automated procedures for domain construction.

The paper is organised as follows. We introduce *meta-syntax*, a variant of BNF in Section 2, and present a collecting semantics for meta-syntax grammars. We apply standard abstract interpretation to underapproximate the collecting semantics of meta-syntax grammars in Section 3. A sub-language is an underapproximation of a language. A sub-language derived by abstract interpretation is an inductively defined sub-language of an inductively defined language. We present our case study in Section 4 and review related work in Section 5.

2 Meta-syntax

In this section, we introduce a variant of BNF called *meta-syntax*, and define the interpretation of meta-syntax grammars using domains and transformers. The difference between BNF and meta-syntax is clarified below.

Example 1. Let *Prop* be a set of propositions and p range over *Prop*. BNF definitions for propositional logic and a sub-logic is given below.

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \qquad \text{Propositional logic}$$
$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \qquad \text{Monotone propositional logic}$$

The first line above can be read as follows: a propositional formula φ is a proposition, or is the composition of formulae using the Boolean connectives $\wedge$, $\vee$ and $\neg$. The grammars above are similar, but not identical. The meta-syntax grammar below specifies a language containing constants and operators.

$$\mathsf{f} ::= \mathsf{prop} \mid \mathsf{op}(\widehat{\mathsf{f}}) \qquad \mathsf{Gram}$$

The two logics above can be obtained by instantiating the macros prop and op as shown below.

$$\mathsf{inst}_1 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\neg, \vee, \wedge\}\} \quad \mathsf{inst}_2 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\vee, \wedge\}\}$$

We prefer to use meta-syntax over BNF because we can instantiate *the same grammar* in different ways to derive different logics. More precisely, a grammar is a structure analogous to a program. The instantiations above can be viewed as specifying two different abstract transformers for the same syntactic macros. We show later that meta-syntax allows us to describe the syntactic relationship between the formulae in monotone propositional logic and propositional logic using abstract interpretation. ◁

The meta-syntax system below syntactically resembles BNF, but is based on Istrail's [15] fixed point characterisation of recursively enumerable languages. Though meta-syntax is *less general* than Cousot and Cousot's bi-inductive semantics [10], we have chosen to work with meta-syntax because it suffices for this paper, and because it is similar to BNF.

Definition 1. *Let* Var *be a set of meta-variables and* Sym*, a set of meta-symbols. A syntax term is one of the following.*

1. *A meta-variable* x *or a meta-symbol* s*.*
2. *The composition* $\mathsf{s}(t_0, \ldots, t_n)$ *of a meta-symbol with terms.*
3. *The uniform composition* $\mathsf{s}(\widehat{t})$ *of a meta-symbol* s *with a term* t*.*
4. *The substitution* $\mathsf{t}_1[\mathsf{x}/\mathsf{t}_2]$ *of a meta-symbol* x *in a term* t_1 *with a term* t_2*.*

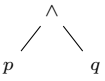
A grammar rule is of the form $\mathsf{x} ::= t_0 \mid \cdots \mid t_n$*, where* t_i *is a syntax term. A grammar is a finite set of grammar rules.*

A meta-variable is *free* in a grammar if it does not occur on the left hand side of a grammar rule. Uniform composition, not present in BNF, is required to collate symbols like $\wedge$ and $\neg$ independent of their arity.

The language of a grammar is analogous to the semantics of a program. The operational semantics of a program statement is given by state transitions, the collecting semantics is defined using state transformers, and properties of programs are characterised by fixed points. Similarly, the meaning of a meta-syntax term is given by a set of syntax trees, the collecting semantics is defined using syntax transformers on a powerset lattice of syntax trees, and languages are defined by fixed points. A formalisation of this idea follows. An alternative formalisation is to use partial-expressions instead of syntax trees.

Signatures, Syntax Trees and Languages. A *signature* (Sig, ar) is a set of symbols Sig with an arity function $ar : Sig \to \mathbb{O}$, associating an ordinal with each symbol. The arity function is left implicit. Ordinals are required for infinitary conjunction and disjunction. A nullary symbol has arity 0. Fix a signature Sig.

A *syntax tree* $\tau = (V, E, sym)$ is a finite-height, ordered tree (V, E) with a function $sym : V \to Sig$ labelling vertices with symbols. The children of a vertex form a possibly infinite sequence $\bar{v} = v_0, v_1, \ldots$. The root of a tree is $root(\tau)$. The set of syntax trees over Sig is $\mathsf{Syn}(Sig)$, written Syn if Sig is clear. A *well formed tree* satisfies that every vertex v has $ar(sym(v))$ children. If we assume the symbol $\wedge$ to have arity 2, the tree on the left is not well-formed but the tree on the right is. An *expression* is a well-formed syntax tree and a *language* is a set of expressions.



An *instantiation* $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ is a set of syntax trees Syn with an instantiation function $\mathsf{inst} : \mathsf{Sym} \to \mathcal{P}(\mathsf{Syn})$ satisfying that, for each s in Sym, $\mathsf{inst}(\mathsf{s})$ contains single vertex trees. (Strictly speaking, vertices are not symbols because operations on symbols and trees differ.) A *syntax environment* (or just environment) $\mathsf{env} : \mathsf{Var} \to \mathcal{P}(\mathsf{Syn})$ maps meta-variables to sets of syntax trees.

The *meaning* of a term t given an instantiation $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ and environment env, is denoted $\|t\|_{\mathsf{In},\mathsf{env}}$ and is defined below. We abbreviate the symbol for the arity of the root symbol of a tree τ to $ars(\tau)$.

$$
\begin{aligned}
\|\mathsf{x}\|_{\mathsf{In},\mathsf{env}} &\hat{=} \mathsf{env}(\mathsf{x}) \\
\|\mathsf{s}(t_0,\ldots,t_{n-1})\|_{\mathsf{In},\mathsf{env}} &\hat{=} \{\tau \mid \mathit{root}(\tau) \text{ is in } \mathsf{inst}(\mathsf{s}), ars(\tau) = n \\
&\qquad \text{and child } \tau_i \text{ of } \mathit{root}(\tau) \text{ is in } \|t_i\|_{\mathsf{In},\mathsf{env}}\} \\
\|\mathsf{s}(\widehat{t})\|_{\mathsf{In},\mathsf{env}} &\hat{=} \{\tau \mid \mathit{root}(\tau) \text{ is in } \mathsf{inst}(\mathsf{s}) \text{ and for } i < ars(\tau) \\
&\qquad \tau \text{ has children } \tau_i \in \|t\|_{\mathsf{In},\mathsf{env}}\} \\
\|t_1[\mathsf{x}/t_2]\|_{\mathsf{In},\mathsf{env}} &\hat{=} \{\tau \mid \tau \text{ is in } \|t_1\|_{\mathsf{In},\mathsf{env}'}, \text{ where} \\
&\qquad \mathsf{env}' \text{ maps } \mathsf{x} \text{ to } \|t_2\|_{\mathsf{In},\mathsf{env}}, \text{ and } \mathsf{y} \neq \mathsf{x} \text{ to } \|\mathsf{y}\|_{\mathsf{In},\mathsf{env}}\} \\
\|t_1 \mid t_2\|_{\mathsf{In},\mathsf{env}} &\hat{=} \|t_1\|_{\mathsf{In},\mathsf{env}} \cup \|t_2\|_{\mathsf{In},\mathsf{env}}
\end{aligned}
$$

The terms above may contain free variables. The *language of a grammar* $\mathsf{x}_1 ::= t_1, \ldots, \mathsf{x}_n ::= t_n$, with no free variables is an environment env, satisfying that $\mathsf{env}(\mathsf{x}_i)$ equals $\|t_i\|_{\mathsf{In},\mathsf{env}}$, for every i.

Domains and Transformers. We now define the semantics of a grammar in terms of domains and transformers. The *domain of syntax trees* is the powerset lattice $(\mathcal{P}(\mathsf{Syn}), \subseteq, \cap, \cup)$ containing sets of syntax trees ordered by inclusion, and closed under union, intersection and set complement. The *domain of instantiation functions* is $(\mathsf{Sym} \to \mathcal{P}(\mathsf{Syn}), \sqsubseteq, \sqcap, \sqcup)$, where the order is defined *pointwise*:

$\mathsf{inst}_1 \sqsubseteq \mathsf{inst}_2$ if $\mathsf{inst}_1(\mathsf{s}) \subseteq \mathsf{inst}_2(\mathsf{s})$ holds for every meta-symbol s.

The *pointwise meet* $\mathsf{inst}_1 \sqcap \mathsf{inst}_2$ of two instantiation functions maps each meta-symbol s to $\mathsf{inst}_1(\mathsf{s}) \cap \mathsf{inst}_2(\mathsf{s})$. The pointwise join is similarly defined. The *domain of syntax-environments* is $(\mathsf{Env}, \sqsubseteq, \sqcap, \sqcup)$, where Env is $\mathsf{Var} \to \mathcal{P}(\mathsf{Syn})$ and the operations are defined pointwise. All these domains are complete lattices. The least element of Env, denoted $\mathsf{env}_\emptyset$, maps all meta-variables to the emptyset.

Fix an instantiation In. We define the transformers for each entity in meta-syntax. The definitions below can be read like the definitions of transformers for statements in a programming language.

A *syntax transformer* $syn_t : \mathsf{Env} \to \mathcal{P}(\mathsf{Syn})$ for a term t maps an environment env to the set of syntax trees $\|t\|_{\mathsf{In},\mathsf{env}}$. For brevity, we do not enumerate the definition of syn_{c}, $syn_{\mathsf{f}(\widehat{c})}$, and $syn_{t_1[\mathsf{x}/t_2]}$, but we emphasise that syn_t is inductively defined. Note that the transformer defined by the separator $\mid$ is union, so $\mid$ the equivalent of a join point in a flow graph. A grammar rule $\mathsf{x} ::= r$ defines a transformer $syn_{\mathsf{x}::=r} : \mathsf{Env} \to \mathsf{Env}$ that maps an environment env to $\mathsf{env}[\mathsf{x}/\, syn_r(\mathsf{env})]$. To define the transformer for a grammar rule, we require a function null to filter out nullary symbols.

$$
\begin{aligned}
\mathsf{null}(\mathsf{s}) &\hat{=} \{\tau \in \mathsf{inst}(\mathsf{s}) \mid ars(\tau) = 0\} & \mathsf{null}(\mathsf{x}) &\hat{=} \emptyset \\
\mathsf{null}(\mathsf{s}(\widehat{t})) &\hat{=} \mathsf{null}(\mathsf{s}) & \mathsf{null}(t_1 \mid t_2) &\hat{=} \mathsf{null}(t_1) \cup \mathsf{null}(t_1)
\end{aligned}
$$

A rule $\mathsf{x}_i ::= r_i$ generates an equation

$$\mathsf{x}_i = \mathsf{null}(r_i) \sqcup \mathit{syn}_{r_i}(\mathsf{env})$$

and a grammar generates a system of equations. Recall that the reachable states of a transition system are the fixed point of an equation $X = \mathit{Init} \cup \mathit{post}(X)$, where *Init* is a set of initial states and *post* is the successor transformer. Observe that the equations above have the same form. The *language of a grammar* is the environment representing the least solution to these equations.

Example 2. Let us derive a fixed point for the grammar and instantiation inst_1 in Example 1. Iterating through the grammar produces the environments below.

$$\mathsf{env}_0 = \{\mathsf{f} \mapsto \emptyset\} \qquad \mathsf{env}_1 = \{\mathsf{f} \mapsto \mathit{Prop}\} \quad \cdots \qquad \mathsf{env}_n = \{\mathsf{f} \mapsto T_n\}$$

The initial environment is empty. After one iteration, we have the set of propositions. After n iterations, T_n contains expressions with at most $n-1$ operators. The fixed point of this sequence maps f to all propositional formulae. ◁

Proposition 1. *The language of a grammar with respect to an instantiation is the least fixed point of the system of equations the grammar generates.*

Semantic Structures. Grammars specify syntax. The semantics associated with the syntax is usually given by an interpretation function, which can be viewed as the operational semantics of the language. The collecting semantics is often derived by lifting the operational semantics to a powerset lattice. We directly assign a "collecting interpretation" to a language. For example, the standard interpretation of $+$ is a binary function in $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The lattice interpretation of $+$ is interpreted as a unary function $\mathcal{P}(\mathbb{N} \times \mathbb{N}) \to \mathcal{P}(\mathbb{N})$ that maps every pair (m, n) in a set X to $m + n$.

A *semantic structure* $\mathcal{S} = (\mathcal{A}, \mathcal{F})$ for a signature *Sig* is a family of lattices $\mathcal{A}$ called *domains* and a family of functions $\mathcal{F}$ called *interpretations*. Each symbol f is interpreted as a function $f^{\mathcal{S}} : A_{f,i} \to A_{f,o}$ with input and output domains $A_{f,i}$ and $A_{f,o}$. If all the domains are identical, we write A for $\mathcal{A}$.

A *structure* $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$ consists of a grammar, an instantiation, and a semantic structure. The *meaning* of an expression in $\mathcal{S}$ is *partially* defined below.

$$\begin{aligned}
[\![c]\!]_{\mathcal{S}} &\hat{=} c^{\mathcal{S}} \text{ where } c, \text{ is a symbol with arity } 0 \\
[\![f(\bar{a})]\!]_{\mathcal{S}} &\hat{=} f^{\mathcal{S}}([\![a_0]\!]_{\mathcal{S}}, \ldots, [\![a_{n-1}]\!]_{\mathcal{S}}), \text{ where } f(\bar{a}) \in \|\mathsf{s}(t_0, \ldots, t_{n-1})\|_{\mathsf{In},\mathsf{env}} \\
[\![f(\bar{a})]\!]_{\mathcal{S}} &\hat{=} f^{\mathcal{S}}([\![a_0]\!]_{\mathcal{S}}, \ldots, [\![a_i]\!]_{\mathcal{S}}, \ldots), \text{ where } f(\bar{a}) \in \|\mathsf{s}(\widehat{t})\|_{\mathsf{In},\mathsf{env}}
\end{aligned}$$

The definitions above suffice because every syntax tree arises from a meta-syntax term. The definition is partial because the arguments of a function symbol must have the same domain as their sub-expressions. It is routine to make this notion precise using types.

3 Abstraction of Syntax

This section contains two ideas. First, we use abstract interpretation to derive underapproximations of the fixed point of a grammar and obtain sub-languages. A grammar defines a language, and every subset of the language is a sub-language. An arbitrary subset may, however, not have an inductive definition related to the grammar. By applying abstract interpretation, we can underapproximate fixed points by fixed points, thereby deriving inductively defined sub-languages of an inductively defined language.

The second idea is to underapproximate syntax to derive overapproximations of semantics. The idea is straightforward, but relies on non-trivial conditions relating abstract domains and logics.

Abstract Interpretation. Let $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$ be posets. Two functions $\alpha : L \to M$ and $\gamma : M \to L$ form a *Galois connection* if,

$$\text{for all } x \in L \text{ and } y \in M,\ \alpha(x) \preccurlyeq y \text{ if and only if } x \sqsubseteq \gamma(y).$$

If L is a powerset lattice, abstractions with respect to $\subseteq$ are *overapproximating* and those with respect to $\supseteq$ are *underapproximating*. Several abstract elements may have the same concretisation. For instance, all intervals $[a, b]$ with a greater than b concretise to the empty set. Such redundancies do not occur in a Galois insertion. A *Galois insertion* is a Galois connection in which γ is injective.

Example 3. This example shows that the standard task of defining a sub-language, such as the negation-free fragment of propositional logic, can be completely formalised and understood as deriving and applying best abstract transformers in abstract interpretation. Recall the grammar and instantiation inst from Example 2. The signature of propositional logic is given by Sig_1 below and of the negation-free fragment by Sig_2 below.

$$Sig_1 = Prop \cup \{\wedge, \vee, \neg\} \qquad Sig_2 = Prop \cup \{\wedge, \vee\}$$

These signatures define domains which form an underapproximating Galois connection with abstraction and concretisation given below.

$$\mathcal{P}(\mathsf{Syn}(Sig_1)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\mathsf{Syn}(Sig_2)) \qquad \alpha(T) = T \cap \mathsf{Syn}(Sig_2) \qquad \gamma(S) = S$$

The abstraction function maps a set of syntax trees to the subset that is negation-free, and the concretisation function is the identity function. The Galois connection lifts pointwise to the instantiation and environment domains.

If we have an instantiation function inst as shown below, we can *derive* the *abstract instantiation* function $\alpha \circ \mathsf{inst} \circ \gamma$ also shown below.

$$\mathsf{inst} = \{\mathsf{prop} \mapsto Prop, \mathsf{op} \mapsto \{\wedge, \vee, \neg\}\}$$
$$\alpha \circ \mathsf{inst} \circ \gamma = \{\mathsf{prop} \mapsto Prop, \mathsf{op} \mapsto \{\wedge, \vee\}\}$$

The abstract instantiation function is the best abstract transformer approximating inst. The fixed point of the grammar for propositional logic with the abstract instantiation defines the set of negation-free formulae. ◁

Example 4. This example illustrates a more complex abstract domain. Let ATree be the set of syntax trees (not necessarily well-formed) with at most one negation symbol. Observe that ATree cannot be of the form $\mathsf{Syn}(Sig)$, because Sig would contain negation and the resulting trees, multiple negations. Nonetheless, $\mathcal{P}(\mathsf{ATree})$ is a subset of $\mathcal{P}(\mathsf{Syn}(Sig_1))$ from Example 3. Intuitively, the *abstract syntax transformers* operate on the lattice $\mathsf{Var} \to \mathcal{P}(\mathsf{ATree})$, and the fixed point of the grammar contains formulae with at most one negation. ◁

Sub-languages by Abstract Interpretation. A *sub-signature* is a subset of a signature. A *sub-language* is a subset of a language. Every underapproximation of the fixed point of a grammar is trivially a sub-language. We are interested in inductively defined sub-languages, because they finitely represent infinitely many formulae. We derive such languages using abstract interpretation to find underapproximations that are also fixed points.

Definition 2. *An instantiation* $\mathsf{AIn} = (\mathsf{ATree}, \mathsf{ainst})$ *is a syntactic abstraction of* $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ *if there is an underapproximating Galois connection* $\mathcal{P}(\mathsf{Syn}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\mathsf{ATree})$ *and the inclusion* $\gamma(\mathsf{ainst}(\mathsf{s})) \subseteq \mathsf{inst}(\mathsf{s})$ *holds for all symbols* s.

Fix a syntactic abstraction $\mathsf{AIn} = (\mathsf{ATree}, \mathsf{ainst})$. We call AIn a *signature abstraction* if ATree contains all trees over a signature. The set ATree in Example 4 does not contain all trees over a signature.

The set of *abstract environments* is $\mathsf{AEnv} = \mathsf{Var} \to \mathcal{P}(\mathsf{ATree})$. The *abstract syntactic transformer* $asyn_t : \mathsf{AEnv} \to \mathcal{P}(\mathsf{ATree})$ for a term t is $\alpha \circ syn_t \circ \gamma$, where α and γ are defined by pointwise lifting. The abstract transformer for a grammar rule is similarly defined, and the abstract equations are obtained by replacing syntax transformers by their abstract counterparts. The *abstract language* defined by a grammar is the least abstract environment that represents a solution to the abstract equations. From standard abstract interpretation theory, we have that the abstract language underapproximates the language of a grammar.

Proposition 2. *The abstract language of a grammar underapproximates the language of a grammar.*

Languages to Domains. We now relate sub-languages to abstract domains over lattices representing semantic objects. A semantic abstraction $(\mathcal{A}, \mathcal{F})$ of a semantic structure $(\mathcal{C}, \mathcal{G})$ satisfies two conditions for every f in Sig.

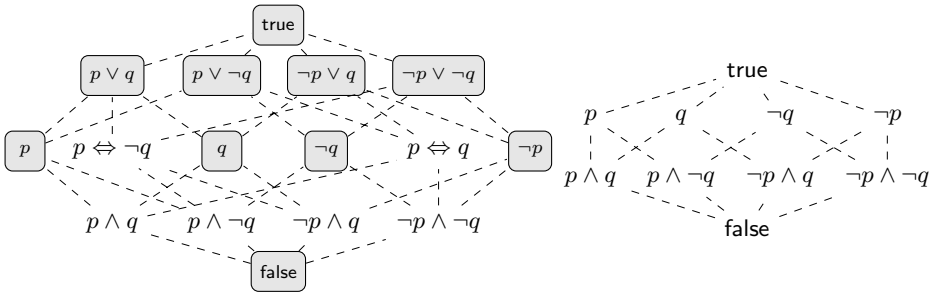
1. There is Galois connection $(C_{f,i}, \alpha_{f,i}, \gamma_{f,i}, A_{f,i})$, and a similar one for f, o.
2. The interpretations satisfy the soundness condition $\alpha_{f,o} \circ f^{\mathcal{C}} \sqsubseteq f^{\mathcal{A}} \circ \alpha_{f,i}$.

Fix a structure $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$ defining a language Lang. Since we interpret languages over lattices, sub-languages define posets. The *structure defined by* Lang,

denoted *struct*(Lang), is the family of posets that together contain the elements $\{[\![e]\!]_{\mathcal{S}} \in \ | \ e \in \mathsf{Lang}\}$, and are ordered as before. The next example illustrates sub-languages *struct*(Lang) and how they define lattices.

Example 5. All languages in this example can be derived by abstraction of syntax, but are presented informally to save space. Let Lang_1 be propositional logic over two variables p and q. The semantic domain is the set of truth assignments $\mathcal{P}(Prop \to \mathbb{B})$. The lattice *struct*($\mathsf{Lang}_1$) is shown on the left below.

Consider a language Lang_2 closed under conjunction, with negation only of propositions. The lattice *struct*(Lang_2) is on the right. There is a Galois insertion from *struct*(Lang_1) to *struct*(Lang_2).



A third language Lang_3 contains propositions, negation of propositions, and is closed under disjunction. The elements of *struct*(Lang_3) are shaded in the left-lattice. There is no Galois connection from *struct*(Lang_1) to *struct*(Lang_3) because *struct*(Lang_3) contains p and q, but not $p \wedge q$. However, Lang_3 is a sub-language of Lang_1 and can be derived by abstraction of syntax. ◁

The poset defined by Lang_3 above does not admit a Galois connection because $p \wedge q$ do not have a unique, minimal overapproximation. We resolve this problem with the following result from lattice theory. Consult the textbook [11, Theorem 7.3] or Cousot and Cousot's result on Moore families [5] for details.

Theorem 1. *An abstract domain A is in a Galois insertion with C exactly if A is a subset of C closed under meets.*

This characterisation allows us to relate languages with abstract domains in the setting of Galois insertions. A language that defines a structure closed under arbitrary meets also defines an abstract domain. Let Lang be a language interpreted over a semantic structure $\mathcal{S} = (\mathcal{A}, \mathcal{F})$. For simplicity, assume that $\mathcal{A}$ is a single domain. The general case is similar to what follows but notationally cumbersome. The language Lang is *completely conjunctive* if for every set of expressions $E \subseteq \mathsf{Lang}$, there exists an expression C_E in Lang satisfying the equality $\bigsqcap \{[\![e]\!]_{\mathcal{S}} \mid e \in E\} = [\![C_E]\!]_{\mathcal{S}}$. The theorem below is new, and states that completely conjunctive languages define abstract domains.

Theorem 2. *Let* Lang *be a language interpreted over a domain* $(C, \sqsubseteq, \sqcap, \sqcup)$. *If* Lang *is completely conjunctive, the poset struct*(Lang) *is a complete lattice that is in a Galois insertion with* C.

The theorem can be proved by showing that for a completely conjunctive Lang, *struct*(Lang) is meet-closed, and then by invoking Theorem 1.

Discussion. We have considered the Galois insertion and complete lattice setting of abstract interpretation for two reasons. We require concretisation functions to be injective because every element of the abstract domain will have only one representation. Abstract domains used in practice may contain redundant representations of the same semantic element. Such domains are not definable in the approach we give above. We consider complete lattices because when arbitrary meet operations are defined, every concrete element has a unique over-approximation. Our completeness assumption excludes important domains such as polyhedra, or regular languages, which do not form complete lattices.

The framework in this paper can be extended to logics whose interpretation defines posets, by interpreting formulae in the logic and using the implication order. In this weaker setting, abstraction functions have to be manually defined, so the abstract element that overapproximates arbitrary concrete elements will not be defined by the framework. Extend this framework to derive abstract domains in which the concretisation function is not injective requires significantly more effort. This is because we map expressions to their meaning, and to obtain redundant representations, we must weaken the meaning function $[\![\cdot]\!]_{\mathcal{S}}$ to distinguish between semantically equivalent formulae.

4 Syntactic Derivation of Semantic Domains

We now follow the approach below to generate specifications of abstract domains.

1. Define a structure $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$, for a logic expressing properties of a program and it's datatypes, and let $\mathcal{S}$ be the semantic structure for this logic.
2. Choose parameters defining subsets of syntax trees.
3. Fix the parameters and compute the sub-language to obtain a sub-logic L.
4. If L is closed under infinitary conjunction, *struct*(L) is an abstract domain.

In this section, we apply this flow to a specific logic and derive many abstract domains existing in practice.

A Logic of Program Properties. The logic we consider is an extension of Presburger arithmetic with infinitary conjunction and disjunction, and with the next-state modality. This logic can encode properties (such as reachability or termination) of programs written in Turing complete languages. The grammar below defines a language with terms and formulae.

$$\mathsf{t} ::= \mathsf{var} \mid \mathsf{fun}(\widehat{\mathsf{t}}), \quad \mathsf{f} ::= \mathsf{pred}(\widehat{\mathsf{t}}) \mid \mathsf{bool}(\widehat{\mathsf{f}}) \mid \mathsf{mod}(\widehat{\mathsf{f}})$$

The signature *Sig* contains variables *Var* and the sets below (k is a positive integer). The binary predicate $\equiv_k$ denotes congruence modulo k. Other symbols have their standard arity.

$$Fun \mathrel{\hat{=}} \{+\} \cup \mathbb{N} \quad Pred \mathrel{\hat{=}} \{<, \leq, =, >, \geq, \equiv_k\} \quad Bool \mathrel{\hat{=}} \left\{\bigvee, \bigwedge, \wedge, \vee, \neg\right\}$$

The instantiation of meta-symbols is below.

$$\mathsf{var} \mapsto Var \quad \mathsf{fun} \mapsto Fun \quad \mathsf{pred} \mapsto Pred \quad \mathsf{bool} \mapsto Bool \quad \mathsf{mod} \mapsto \{\mathsf{EX}\}$$

An example formula is $\mathsf{EX}(x = y + y)$, stating that there is a successor state in which x equals $2y$. Familiar modalities can be recovered using infinitary operators: $\mathsf{AX}\varphi$ is the formula $\neg\mathsf{EX}\neg\varphi$, which is satisfied by a state if all its successors satisfy φ. Let $\mathsf{AX}^i\varphi$ denote the formula $\mathsf{AX}\cdots\mathsf{AX}\varphi$, in which AX occurs i-times in sequence. The formula $\mathsf{AG}\varphi$, defined as $\bigwedge_i \mathsf{AX}^i\varphi$, asserts that φ is true on every path. Infinitary operators can also be used to express multiplication.

We interpret this logic over transition systems. Let *Val* be a set of values and $Env \mathrel{\hat{=}} Var \to Val$ be the set of states (program environments). A *transition system* $M = (Env, E)$ contains a relation $E \subseteq Env \times Env$. Recall that E defines a *predecessor transformer* $pre : \mathcal{P}(Env) \to \mathcal{P}(Env)$ as below.

$$pre(X) \mathrel{\hat{=}} \{s \in Env \mid \text{ there exists } t \text{ in } X \text{ and } (s,t) \text{ is in } E\}$$

The semantic structure we need contains the lattices $\mathcal{P}(\mathbb{N})$ and $\mathcal{P}(Env)$. The semantics of a term t is a set of values $[\![t]\!]_{\mathcal{S}} : \mathcal{P}(Env) \to \mathcal{P}(\mathbb{N})$, and of a formula φ is a set of environments $[\![\varphi]\!]_{\mathcal{S}} : \mathcal{P}(Env) \to \mathcal{P}(Env)$, both defined below.

$$[\![x]\!]_{\mathcal{S}} \mathrel{\hat{=}} X \mapsto \{\varepsilon(x) \mid \varepsilon \in X\}$$
$$[\![t_1 + t_2]\!]_{\mathcal{S}} \mathrel{\hat{=}} X \mapsto \{m + n \mid m \in [\![t_1]\!]_{\mathcal{S}}, n \in [\![t_2]\!]_{\mathcal{S}}\}$$

$$[\![\mathsf{EX}\varphi]\!]_{\mathcal{S}} \mathrel{\hat{=}} pre([\![\varphi]\!]_{\mathcal{S}}) \qquad [\![P(\bar{t})]\!]_{\mathcal{S}} \mathrel{\hat{=}} \{\varepsilon \in Env \mid \varepsilon \text{ satisfies } P(\bar{t})\}$$
$$[\![\varphi \wedge \psi]\!]_{\mathcal{S}} \mathrel{\hat{=}} [\![\varphi]\!]_{\mathcal{S}} \cap [\![\psi]\!]_{\mathcal{S}} \qquad [\![\varphi \vee \psi]\!]_{\mathcal{S}} \mathrel{\hat{=}} [\![\varphi]\!]_{\mathcal{S}} \cup [\![\psi]\!]_{\mathcal{S}}$$

Negation is interpreted as set-complement and infinitary conjunction and disjunction are arbitrary union and intersection, respectively. We now derive abstractions of $\mathcal{P}(Env)$ using abstraction of syntax.

Sub-language Parameters. In the previous section, we showed how one can generate a sub-language by starting with a set of syntax trees. We use a notion of parameters to generate syntax trees.

A *k-variable predicate* $P(\bar{t})$ is one that contains at most k variables. A *syntax parameter* is a tuple (S, k) consisting of a signature $S \subseteq Sig$ and a value k from $\mathbb{N} \cup \{\infty\}$. A *syntax parameter* (S, k) defines a set of syntax trees ATree over the symbols $S \cup Var \cup \{\bigwedge, \wedge\}$ and containing only k-variable predicates. Note that we always include variables, finite and infinitary conjunction in every syntax abstraction. Define an order $(Q, m) \sqsubseteq (S, n)$ that holds if $Q \subseteq S$ and $m \leq n$

both hold. The parameter order implies subset inclusion of the syntax trees they represent.

The results of the previous section imply that evaluating the grammar Gram over ATree defines a sub-language Lang for every parameter value, and that *struct*(Lang) defines an abstract domain over $\mathcal{P}(Env)$. Let *struct*(P) denote the substructure for the language generated by a parameter P. We will illustrate the passage from parameters, via sub-languages to abstract domains. To reduce clutter, we write the parameter $(\{\leq, \equiv_2\}, 3)$ as $(\leq, \equiv_2; 3)$ in figures.

Example 6. Consider the transition system M, on the left below. States represent the values of a variable x. A standard method of abstraction is to partition states using predicates. The result N of partitioning states of M using the predicates $x > 0$ and $x = 0$ is on the right below.

We will see how this abstraction, and others, can be derived syntactically. Three parameters are given below and satisfy the order $P_3 \sqsubseteq P_2 \sqsubseteq P_1$.

$$P_1 \hat{=} (\{=0, >0, \vee, \neg, \mathsf{EX}\}, 1)$$
$$P_2 \hat{=} (\{=0, >0, \vee, \mathsf{EX}\}, 1)$$
$$P_3 \hat{=} (\{=0, >0, \mathsf{EX}\}, 1)$$

The language generated by P_1 contains one place predicates for equality with 0 and being strictly greater than 0, is closed under Boolean operations, and has predicates with at most one variable. The predicate $x < 0$ is expressible in this language but $x < y$ is not. The language generated by P_2 is only closed under conjunction and disjunction but not negation, while the language generated by P_3 is only closed under conjunction. All the languages are also closed under the modal operator EX.

The structures *struct*(P_i) for each parameter contain a lattice representing the semantics of predicates and a transformer for the semantics of EX. These substructures are shown in Figure 2. We only present the calculation of formulae in *struct*(P_3). The elements generated by formulae in P_1 and P_2 can be derived by Boolean combinations of elements in *struct*(P_3). We show a formula on the left and its interpretation in Figure 2 on the right. Assume that formulae are interpreted over the transition system M, with EX interpreted as the predecessor operator.

$$[\![x = 0]\!] = (x = 0) \qquad [\![x > 0]\!] = (x > 0) \qquad [\![\neg x > 0]\!] = (x \leq 0)$$
$$[\![\mathsf{EX}\ x = 0]\!] = (x \leq 0) \qquad [\![\mathsf{EX}\ x > 0]\!] = (x \leq 0) \qquad [\![\neg x = 0]\!] = (x \neq 0)$$

Every predicate in the first lattice represents a set of states of N above and *pre* is the predecessor function for N. Note that this abstraction *was not* derived

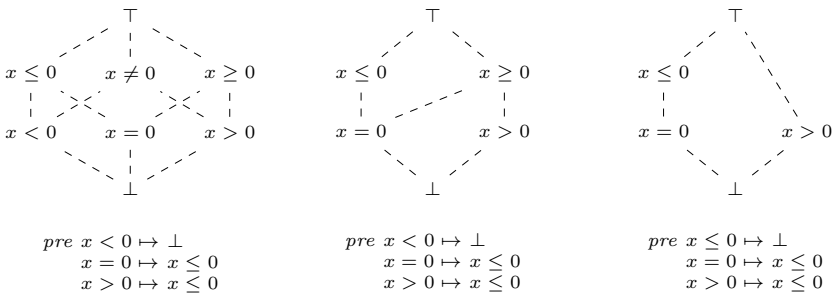


Fig. 2. Abstract domains generated signature abstraction

from N but by evaluating formulae. The other two lattices are not closed under Boolean operations, hence cannot be represented as transition systems. ◁

The set of syntax parameters with the order $\sqsubseteq$ forms a lattice, and each element of this lattice defines an abstract domain. The lattice of syntax parameters can be viewed as a lattice of abstract domains in which $(Q, m) \sqsubseteq (S, n)$ implies that the abstract domain defined by (S, n) refines the domain defined by (Q, m). In fact, the lattice of syntax parameters is an abstraction of the lattice of abstract domains [5].

Generating Abstract Domains. We now show that appropriate parameters generate abstract domains that are used in practice. We emphasise again that this approach does not solve all the issues that arise in designing an abstraction. Nonetheless, it provides an alternative, syntax-based approach, which we believe is more amenable to automation. We now show that a small range of parameter values generates the specifications of many abstract domains used in practice.

Relational and Non-relational Abstractions. A standard form of abstraction is to decouple the relationship between variables. Non-relational domains cannot express facts like $x = y$, while relational abstractions can. The terms weakly-relational refers to domains that express a limited amount of relational information. The octagon domain is weakly relational because it can express $x + y \leq c$, but not $x + y + z \leq c$, because at most two variables can occur in a constraint. A parameter satisfying $(S, m) \sqsubseteq (Sig, 1)$ contains at most one variable per predicate and cannot express relational information. A parameter satisfying $(S, m) \sqsubseteq (Sig, n)$ for a finite, positive integer n generates a weakly relational domain, because some, but not all relational information can be expressed. A parameter (S, ∞) contains predicates with no bound on the number of variables. Such parameters generate domains encoding relational information.

Temporal Abstractions. We use the term temporal abstractions for those based in bisimulation, simulation and similar preorders defined on transition systems.

The facts we use about such equivalences are summarised in [2]. Recall that a relation $R \subseteq Env \times Env$ is a simulation if whenever (s,t) is in R and (s,s') is a transition, there must be a transition (t,t') such that (s',t') is in R. It is known that bisimulation can be characterised by a logic that is closed under Boolean operations, infinitary conjunction, and EX, and that simulation is characterised by a logic closed under infinitary conjunction, disjunction and EX. We can use these results to specify abstract domains based on bisimulation and simulation.

Every parameter greater than $(\{\bigvee, \neg, \mathsf{EX}\}, 0)$ represents a bisimulation quotient with respect to a given set of predicates. The first abstraction in Figure 2 represents the bisimulation quotient of M using the predicates $x < 0$, $x = 0$ and $x > 0$. Every parameter greater than $(\{\bigvee, \mathsf{EX}\}, 0)$, containing infinitary disjunction but not necessarily negation, represents a simulation preserving domain. Such domains are studied in [19].

Numeric Abstractions. We now discuss parameter settings that generate standard abstract domains. For complete rigour, each claim that follows should be accompanied with a proof that the generated domain represents the claimed, existing domain.

- *Affine Equalities* $(\{+, \mathbb{N}, =\}, \infty)$. This abstraction contains conjunctions of constraints of the form $a_1 x_1 + \cdots a_n x_n = k$.
- *Affine Congruences* $(\{+, \mathbb{N}, \equiv_k\}, \infty)$. This abstraction contains conjunctions of constraints of the form $a_1 x_1 + \cdots a_n x_n \equiv_k m$, stating that the constraint on the left is congruent to m, modulo k.
- *Intervals* $(\{\mathbb{N}, \leq\}, 1)$. If at most one variable per predicate is allowed when using inequalities, we have constraints of the form $0 \leq x \wedge x \leq 3$, which encode that x is in the interval $[0, 3]$.
- *Constants* $(\{\mathbb{N}, =\}, 1)$. If at most one variable per predicate is allowed when using equalities, we have constraints of the form $x = 3$. Since no two distinct equalities are satisfiable, all conjunctions of formulae become false. This domain is used in constant propagation.
- *Parity* $(\{0, 1, \equiv_2\}, 1)$. If the domain is non-relational and the only predicate is $\equiv_2$, we obtain the parity domain.
- *Signs* $(\{0, <, =, >\}, 1)$. A non-relational domain in which every variable can only be compared with 0 is the signs domain. The parameter above only generates the 5 element signs domain containing the predicates $x < 0$, $x = 0$ and $x > 0$. If extended with disjunction, we obtain the 8 element signs domain closed under Boolean operations.

We emphasise that a signature can contain infinitely many symbols, and the resulting domain, infinitely many elements. Several numeric domains mentioned above have infinitely many elements. Note also that the domains of constants, affine equalities and congruences, all contain infinitely many inequalities but have finite height. Logically, such domains correspond to logics in which there are no infinite sequences of strict implications. For these reasons, generating domains by abstraction of syntax is difficult to automate. Predicate abstraction [1] applies only to finite sets of predicates, but is automatic. The parameters for some domains we discussed are shown in Figure 3.

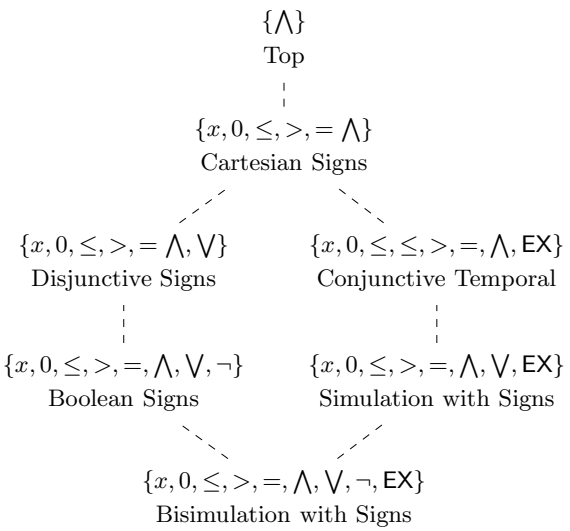


Fig. 3. A lattice of signatures that generates the abstract domains shown

5 Related Work

The ideas that domains can be viewed as logics, and manipulation of grammars can be understood as underapproximation is folklore in the abstract interpretation community. This paper has attempted to formalise this view, and to combine two folklore ideas to obtain a new approach to deriving abstract domains.

The technical background for this paper was drawn from language theory, abstract interpretation, and lattice theory. The language-theoretic basis of our work is the Chomsky-Schützenberger theorem, which characterises context free languages as fixed points. Ginsburg and Rice [14] extended this characterisation to recursively enumerable languages. Istrail [15] showed that concatenation, union, intersection and substitution on languages suffice for the fixed point characterisation. It follows that the language of a BNF grammar has a fixed point characterisation. See Cousot and Cousot [6,7] or Paulson [18], for related discussions of inductive definitions.

We use abstract interpretation to approximate the fixed point defined by a BNF grammar. Cousot and Cousot's bi-inductive semantics [10] has similar motivations with important differences. If we only consider the languages that can be derived, bi-inductive semantics uses posets and possibly non-monotone functions, hence is a strictly more general framework than ours. However, we use BNF grammars, which is how logics are specified in practice. There is much work on grammar abstractions [3,8,17,22], with a focus on program analysis. Our focus is language generation from BNF, akin to invariant generation from a program, while the cited methods above are similar to analysis of a transition system.

Abstract interpretation has also been combined with parsing algorithms [9,21]. Parsing is a language recognition, not language generation task.

6 Conclusion

Approximation of semantics, as formalised by abstract interpretation, is fundamental to tractable reasoning about programs. Abstract reasoning depends on manually defined artefacts such as abstract domains. We have studied the problem of developing a systematic approach to the narrow task of specifying the syntax of an abstract domain. The solution proposed in this paper was to use meta-syntax, a variant of BNF-style grammars to specify abstract domains. Our contribution is to show that abstraction of grammars provides a formal framework for generating specifications of abstract domains. We demonstrated this framework with a simple case study that covered a broad range of abstractions.

The method in this paper is a first step to deriving a systematic framework for thinking about and manipulating abstract domains. We have presented no algorithms for generically deriving implementations of abstract domains. Most numeric abstractions we discussed are sub-logics of Presburger arithmetic, which is decidable. One question is whether implementations of infinite domains over sub-logics of Presburger arithmetic can be derived automatically. Specifically, given a decision procedure for Presburger arithmetic, and a syntactic restriction of formulae can we automatically synthesise abstract transformers for domains such as intervals or constants.

Another problem in dire need of a rigorous framework is that of designing operators to ensure convergence of fixed point iterations. There are results showing that convergence operators cannot be monotone in general. We have shown that abstraction of syntax, in certain situations yields overapproximating abstractions. Another interpretation of this result, is that there are situations in which a syntactically defined subset of a lattice only admits a non-monotone abstraction function. A question arising from our work is whether abstraction of syntax can be used to define convergence acceleration operations. We observe that several standard interval widening operators can be defined in this manner. Investigating such questions further is future work.

Acknowledgements. We thank the reviewers for their careful reading and positive feedback.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
2. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)

3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252. ACM Press (1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages, pp. 269–282. ACM Press (1979)
6. Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretations. In: Principles of Programming Languages, pp. 83–94. ACM Press (1992)
7. Cousot, P., Cousot, R.: Compositional and Inductive Semantic Definitions in Fixpoint, Equational, Constraint, Closure-condition, Rule-based and Game-Theoretic Form. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 293–308. Springer, Heidelberg (1995)
8. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Functional Programming Languages and Computer Architecture, June 25–28, pp. 170–181. ACM Press (1995)
9. Cousot, P., Cousot, R.: Grammar Analysis and Parsing by Abstract Interpretation. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 175–200. Springer, Heidelberg (2007)
10. Cousot, P., Cousot, R.: Bi-inductive structural semantics. Information and Computation 207, 258–283 (2009)
11. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge University Press, Cambridge (1990)
12. Filé, G., Giacobazzi, R., Ranzato, F.: A unifying view of abstract domain design. ACM Computing Surveys 28(2), 333–336 (1996)
13. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. Journal of the ACM 47(2), 361–416 (2000)
14. Ginsburg, S., Rice, H.G.: Two families of languages related to Algol. Journal of the ACM 9, 350–371 (1962)
15. Istrail, S.: Generalization of the Ginsburg-Rice Schützenberger fixed-point theorem for context-sensitive and recursive-enumerable languages. Theoretical Computer Science 18, 333–341 (1982)
16. Jensen, T.P.: Strictness Analysis in Logical Form. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 352–366. Springer, Heidelberg (1991)
17. Okhotin, A., Reitwießner, C.: Conjunctive grammars with restricted disjunction. Theoretical Computer Science 411, 2559–2571 (2010)
18. Paulson, L.C.: A Fixedpoint Approach to Implementing (Co)Inductive Definitions. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 148–161. Springer, Heidelberg (1994)
19. Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. of Logic and Computation 17(1), 157–197 (2007)
20. Schmidt, D.A.: Internal and External Logics of Abstract Interpretations. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 263–278. Springer, Heidelberg (2008)
21. Schmitz, S.: Approximating Context-Free Grammars for Parsing and Verification. Thèse de doctorat, Université de Nice-Sophia Antipolis, France (2007)
22. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. ACM Transactions on Software Engineering Methodology 16(4), 14 (2007)

Logico-Numerical Max-Strategy Iteration*

Peter Schrammel[1] and Pavle Subotic[2,3]

[1] Inria Grenoble – Rhône-Alpes, France
peter.schrammel@inria.fr
[2] Uppsala University, Sweden
[3] University of Sydney, Australia
pavle.subotic@it.uu.se

Abstract. Strategy iteration methods are used for solving fixed point equations. It has been shown that they improve precision in static analysis based on abstract interpretation and template abstract domains, *e.g.* intervals, octagons or template polyhedra. However, they are limited to numerical programs. In this paper, we propose a method for applying max-strategy iteration to logico-numerical programs, *i.e.* programs with numerical and Boolean variables, without explicitly enumerating the Boolean state space. The method is optimal in the sense that it computes the *least fixed point* w.r.t. the abstract domain; in particular, it does not resort to widening. Moreover, we give experimental evidence about the efficiency and precision of the approach.

Keywords: Verification, Static Analysis, Abstract Interpretation, Strategy Iteration.

1 Introduction

This paper deals with the *verification of safety properties* about *logico-numerical* programs, *i.e.*, programs manipulating Boolean and numerical variables. Verification of such properties amounts to checking whether the *reachable state space* is contained in the *invariant* specified by the property.

Classical applications are safety-critical controllers as found in modern transport systems. In such systems the properties to be proved depend on the relationships between the numerical variables of the system. The Boolean state space can be large, especially when analyzing programs written in data-flow languages like Lustre [1] and Scade[1] where the control structure is encoded in Boolean (or finite-type) variables.

Abstract Interpretation. The reachability problem is undecidable for this class of programs, so analysis methods are not complete. Abstract interpretation [2] is a classical method with guaranteed termination for the price of an approximate analysis result. The key idea is to over-approximate sets of states

* This work was partly supported by the Inria large-scale initiative Synchronics.
[1] www.esterel-technologies.com

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 414–433, 2013.

S by elements $S^\sharp$ of an *abstract domain*. A classical abstract domain for numerical invariants in $\wp(\mathbb{R}^n)$ is the domain of convex polyhedra $Pol(\mathbb{R}^n)$ [3]. An approximation $S^\sharp$ of the reachable set S is then computed by iteratively solving the fixed point equation $S = S_0 \cup \mathsf{post}(S)$ *in the abstract domain*. To ensure termination when the abstract domain contains infinitely ascending chains, one applies an extrapolation operator called *widening*, which induces additional approximations.

Strategy Iteration. Strategy (or policy) iteration methods [4–9] are a way to solve the above fixed point equation *without* the need for a widening operator. The main idea of these methods is to iteratively approximate the least fixed point of $S = F(S)$ by fixed points of "simpler", more efficiently computable semantic equations $S = F^{(i)}(S)$, induced by so-called strategies, such that a fixed point of F is guaranteed to be found after a finite number of iterations. Depending on whether the least fixed point is approached from above or below, the methods are called min- or max-strategy iteration respectively.

These techniques are applied to template domains, *i.e.*, abstract domains with *a priori* fixed constraints for which constant bounds are determined during the analysis. Linear templates generate so-called *template polyhedra* [10], which subsume classical domains like intervals [11], zones [12] and octagons [13]. However, these methods are restricted to numerical programs.

Handling Boolean Variables. The difficulty in dealing with logico-numerical programs is that Boolean and numerical variables tightly interact in their evolution.

The classical method to handle Boolean variables in an abstract-interpretation-based analysis is to *explicitly* unfold the Boolean control structure by *enumerating* the Boolean state space and to analyze the numerical variables on the obtained control flow graph using a numerical abstract domain: however, this raises the problem that the analysis becomes intractable already for small to medium-sized programs because the number of control locations grows exponentially with the number of Boolean variables.

Therefore we have to consider an *implicit*, *i.e.* symbolic, treatment of Booleans:

- A naive approach is to encode Booleans as *integers* $\in \{0, 1\}$. The advantage is that max-strategy iteration can be used "as is" by adding template constraints for those Booleans. Though, such an analysis will yield very rough approximations because commonly used abstract domains can only describe convex sets, whereas Boolean state sets are usually highly non-convex.
- *Logico-numerical abstract domains* aim at abstracting state sets of the form $\wp(\mathbb{B}^p \times \mathbb{R}^n)$. One way of representing such sets stems from composite model checking [14] which combines BDDs and Presburger formulas. Domains combining BDDs and numerical abstract domains like polyhedra have been proposed for example by Jeannet et al [15] and Blanchet et al [16].
- Other (not abstract-interpretation-based) approaches rely on *SAT-modulo-theory* solvers, for example the k-induction-based verification tool KIND [17].

In this paper we follow the approach of using logico-numerical abstract domains.

Contributions. Our contributions can be summarized as follows:

1. We describe a novel method for computing the set of reachable states of a logico-numerical program based on max-strategy iteration that avoids the enumeration of the Boolean state space. The technique interleaves truncated Kleene iterations in a logico-numerical abstract domain and numerical max-strategy iterations. The method is optimal, *i.e.* it computes the *least* fixed point w.r.t. the abstract semantics.
2. We give the results of an experimental evaluation: We show that our logico-numerical max-strategy iteration gains one order of magnitude in terms of efficiency in comparison to the purely numerical approach while being almost as precise. Moreover, these are the first experimental results of applying max-strategy iteration to larger programs.

Organisation of the Article. §2 gives an introduction to abstract interpretation with template domains and max-strategy iteration. §3 describes the main contribution, the logico-numerical max-strategy iteration algorithm. §4 presents experimental results, and finally §5 concludes.

2 Preliminaries

Program Model. We consider programs modeled as symbolic control flow graphs over a state space Σ:

Definition 1. *A symbolic control flow graph (CFG) is a directed graph* $\langle L, \mathcal{R}, \ell_0 \rangle$ *where*

- L *is a finite set of locations,* $\ell_0 \in L$ *is the initial location, and*
- $(\ell, R, \ell') \in \mathcal{R}$ *define a finite number of arcs from locations* $\ell \in L$ *to* $\ell' \in L$ *with transition relations* $R \subseteq \Sigma^2$.

An execution of a CFG is a possibly infinite sequence

$$(\ell_0, \boldsymbol{s}_0) \to^R (\ell_1, \boldsymbol{s}_1) \to \dots$$

where $\forall k \geq 0$

- $\ell_k \in L, \boldsymbol{s}_k \in \Sigma$
- $(\ell_k, \boldsymbol{s}_k) \to^R (\ell_{k+1}, \boldsymbol{s}_{k+1})$ if $R(\boldsymbol{s}_k, \boldsymbol{s}_{k+1})$

In the case of affine programs, $\Sigma = \mathbb{R}^n$ and the relations $R(\boldsymbol{x}, \boldsymbol{x'})$ are conjunctions of linear inequalities. Fig. 1 in §2.2 shows the CFG of an affine program.

The methods presented in this paper will focus on logico-numerical programs with $\Sigma = \mathbb{B}^p \times \mathbb{R}^n$ with state vectors $\boldsymbol{s} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{x} \end{pmatrix} \in \Sigma$ and relations $R(\boldsymbol{s}, \boldsymbol{s'})$ with affine, numerical subrelations.

Example 1. An example for such a logico-numerical program is the following C program:

```
b1=true; b2=true; x=0;
while(true)
{
  while(x<=19) { x = b1 ? x+1 : x-1; }
  while(x<=99) { x = b2 ? x+1 : x; b2 = !b2; }
  if (x>=100) { b1 = (x<=100); x = x-100; }
}
```

Fig. 4 in §3.2 shows a CFG corresponding to this program. Note that we allow numerical constraints in assignments to Boolean variables.

A program property we want to prove is for instance the invariant $0 \leq x \leq 100$.

2.1 Abstract Interpretation with Template Polyhedra

Template polyhedra [10] are polyhedra the shape of which is fixed by a so-called template. The domain operations can be performed efficiently with the help of linear programming (LP) solvers.

We will use the following notations: $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$; the operators min, sup, $\leq$, etc are point-wisely lifted to vectors.

Template Polyhedra Abstract Domain. A template polyhedron is generated by a template constraint matrix, or short template, $\mathbf{T} \in \mathbb{R}^{m \times n}$ of which each row contains at least one non-zero entry.

For example, $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ is a template constraint matrix of intervals for a system with a single variable x. It represents the constraints $x \leq d_1 \wedge -x \leq d_2$, *i.e.* $x \in [-d_2, d_1]$.

The set of template polyhedra $\mathcal{P}^{\mathbf{T}}$ generated by $\mathbf{T}$ is $\{X_{\boldsymbol{d}} \mid \boldsymbol{d} \in \overline{\mathbb{R}}^m\}$ with $X_{\boldsymbol{d}} = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{R}^n, \mathbf{T}\boldsymbol{x} \leq \boldsymbol{d}\}$.

An abstract value is represented by the vector of bounds $\boldsymbol{d}$. The analysis tries to find the smallest values of $\boldsymbol{d}$ representing a fixed point of the semantic equations. $\top$ and $\bot$ are naturally represented by the bound vectors $\boldsymbol{\infty}$ and $-\boldsymbol{\infty}$ respectively.

We state the definitions of the domain operations[2]:

- Concretization: $\gamma^x_{\mathbf{T}}(\boldsymbol{d}) = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{R}^n, \mathbf{T}\boldsymbol{x} \leq \boldsymbol{d}\},\ \boldsymbol{d} \in \overline{\mathbb{R}}^m$
- Abstraction: $\alpha^x_{\mathbf{T}}(X) = \min\{\boldsymbol{d} \in \overline{\mathbb{R}}^m \mid \gamma^x_{\mathbf{T}}(\boldsymbol{d}) \supseteq X\},\ X \subseteq \mathbb{R}^n$
- Join: $\boldsymbol{d} \sqcup^x_{\mathbf{T}} \boldsymbol{d}' = \big(\max(d_1, d'_1), \ldots, \max(d_m, d'_m)\big)$
- Image: The templates may vary from location to location: let us denote $\mathbf{T}_\ell$ the template in location ℓ and $\boldsymbol{d}_\ell$ the corresponding vector of bounds. Then the post-image by a transition relation R of transition (ℓ, R, ℓ') in a CFG is defined as follows:

[2] The superscript ^{x} is used to distinguish the (numerical) template polyhedra operations from the logico-numerical domain operations that we are going to define in §3.1.

$$[\![R]\!]^\sharp(\boldsymbol{d}_\ell) = \sup\{\mathbf{T}_{\ell'}\boldsymbol{x}' \mid \mathbf{T}_\ell\boldsymbol{x} \leq \boldsymbol{d}_\ell \ \wedge \ R(\boldsymbol{x},\boldsymbol{x}')\}$$

i.e., given the bounds $\boldsymbol{d}_\ell$ corresponding to the template $\mathbf{T}_\ell$ it returns the bounds corresponding to $\mathbf{T}_{\ell'}$.

Reachability Analysis. Let $R^j_{\ell,\ell'}$ denote a transition relation in the set of transitions from ℓ to ℓ'. We will view $\boldsymbol{d}$ alternatively as the concatenated vector of the bound vectors of all locations and the map $L \to \overline{\mathbb{R}}^m$ which assigns a vector of bounds $\boldsymbol{d}_\ell$ to each location ℓ: $\boldsymbol{d}(\ell) = \boldsymbol{d}_\ell$. $\boldsymbol{d}^0 = \lambda\ell. \begin{cases} \infty \text{ for } \ell = \ell_0 \\ -\infty \text{ for } \ell \neq \ell_0 \end{cases}$ denotes the initial values of the bounds. The set of abstract reachable states represented by the bounds $\boldsymbol{d}$ is computed by:

$$\mathit{lfp}\ \lambda\boldsymbol{d}.\big(\boldsymbol{d}^0 \sqcup^x \lambda\ell'. \bigsqcup_{R^j_{\ell,\ell'}\in\mathcal{R}} [\![R^j_{\ell,\ell'}]\!]^\sharp(\boldsymbol{d}_\ell)\big)$$

2.2 Max-Strategy Iteration

Max-strategy iteration [7, 8, 18–20] is a method for computing the least solution of a system of equations $\mathcal{M}$ of the form $\boldsymbol{\delta} = \boldsymbol{F}(\boldsymbol{\delta})$, where $\boldsymbol{\delta}$ are the template bounds, and F_i, $0 \leq i \leq n$ is a finite maximum of monotonic and concave operators $\mathbb{R}^n \to \mathbb{R}$; in our case they are affine functions. The max-strategy improvement algorithm for affine programs is guaranteed to compute the least fixed point of $\boldsymbol{F}$, and it has to perform at most exponentially many improvement steps, each of which takes polynomial time.

Semantic Equations. The equation system $\mathcal{M}$ is constructed from the abstract semantics of the program's transitions:

$$\text{for each } \ell' \in L : \boldsymbol{\delta}_{\ell'} = \max\Big(\{\boldsymbol{d}^0_{\ell'}\} \cup \{[\![R]\!]^\sharp(\boldsymbol{\delta}_\ell) \mid (\ell, R, \ell') \in \mathcal{R}\}\Big)$$

with $\boldsymbol{d}^0$ and $[\![R]\!]^\sharp(\boldsymbol{d}_\ell)$ as defined above in §2.1.

Note that we use $\boldsymbol{\delta}$ for denoting the vector of bound variables appearing in syntactic expressions, and $\boldsymbol{d}$ for the vector carrying the actual bounds. $\delta_{\ell,i}$ is the bound variable corresponding to the i^{th} line of the template in location ℓ.

Example 2 (Semantic equations). Using the template constraints $\begin{pmatrix} 1 \\ -1 \end{pmatrix} x \leq \begin{pmatrix} d_{\ell,1} \\ d_{\ell,2} \end{pmatrix}$ in locations ℓ, the equation system for location ℓ_1 of the example in Fig. 1 consists of one equation for each template bound variable of which the arguments of the max operator are the initial value $-\infty$ and one expression $[\![R]\!]^\sharp$ for each of the three incoming arcs:

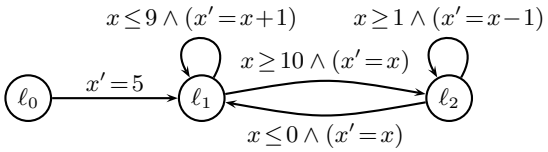


Fig. 1. CFG of an affine program

$$\delta_{1,1} = \max \left\{ \begin{array}{l} -\infty, \\ \sup\{\ x' \mid x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 5 \qquad\qquad\ \}, \\ \sup\{\ x' \mid x \leq \delta_{1,1} \wedge -x \leq \delta_{1,2} \wedge\ x \leq 9\ \wedge x' = x+1\ \}, \\ \sup\{\ x' \mid x \leq \delta_{2,1} \wedge -x \leq \delta_{2,2} \wedge\ x \leq 0\ \wedge\ \ x' = x\ \ \} \end{array} \right\}$$

$$\delta_{1,2} = \max \left\{ \begin{array}{l} -\infty, \\ \sup\{\ -x' \mid x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 5 \qquad\qquad\ \}, \\ \sup\{\ -x' \mid x \leq \delta_{1,1} \wedge -x \leq \delta_{1,2} \wedge\ x \leq 9\ \wedge x' = x+1\ \}, \\ \sup\{\ -x' \mid x \leq \delta_{2,1} \wedge -x \leq \delta_{2,2} \wedge\ x \leq 0\ \wedge\ \ x' = x\ \ \} \end{array} \right\}$$

Strategies. A *strategy* μ induces a "subsystem" $\boldsymbol{\delta} = \widehat{\boldsymbol{F}}(\boldsymbol{\delta})$ of $\mathcal{M}$ in the sense that exactly one argument $\widehat{F}_i$ of the max operator on the right-hand side of each equation $\delta_i = \max(\ldots, \widehat{F}_i(\boldsymbol{\delta}), \ldots)$ is chosen. Intuitively, this means that a strategy selects exactly one "incoming transition" for each template bound variable in each location ℓ'.

Example 3 (Strategy). A strategy in the example in Fig. 1 corresponds for instance the following system of equations:

$$\begin{array}{l} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = \sup\{\ \ x' \mid x \leq \delta_{1,1}\ \wedge\ -x \leq \delta_{1,2}\ \wedge\ x \leq 9\ \wedge x' = x+1\ \} \\ \delta_{1,2} = \sup\{\ -x' \mid x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 5 \qquad\qquad\ \} \\ \delta_{2,1} = -\infty \\ \delta_{2,2} = -\infty \end{array}$$

We see that for $\delta_{1,1}$ we have chosen the third and for $\delta_{1,2}$ the second argument of the max operators in the equations of Example 2.

One has to compute the least solution $\mathit{lfp}[\![\mathcal{M}]\!]$ of the system $\mathcal{M}$, where $[\![\mathcal{M}]\!]$ is defined as

$$[\![\mathcal{M}]\!](\boldsymbol{d}) = \max_{\mu \text{ in } \mathcal{M}} [\![\mu]\!](\boldsymbol{d})$$

and with $[\![\mu]\!](\boldsymbol{d}) = [\![\boldsymbol{\delta} = \widehat{\boldsymbol{F}}(\boldsymbol{\delta})]\!](\boldsymbol{d}) = \widehat{\boldsymbol{F}}(\boldsymbol{d})$.

Max-Strategy Improvement. $\mathit{lfp}[\![\mathcal{M}]\!]$ is computed with the help of the max-strategy improvement algorithm (see Fig. 2) which iteratively improves strategies μ until the least fixed point $\mathit{lfp}[\![\mu]\!]$ of a strategy equals $\mathit{lfp}[\![\mathcal{M}]\!]$.

initial strategy: $\mu := \{\boldsymbol{\delta}_{\ell_0} = \boldsymbol{\infty},\ \boldsymbol{\delta}_\ell = -\boldsymbol{\infty} \text{ for all } \ell \neq \ell_0\}$
initial bounds: $\boldsymbol{d} := \lambda\ell.\boldsymbol{\delta}_\ell \to \begin{cases} \boldsymbol{\infty} & \text{for } \ell = \ell_0 \\ -\boldsymbol{\infty} & \text{for } \ell \neq \ell_0 \end{cases}$
while not $\boldsymbol{d}$ is a solution of $\mathcal{M}$ do
  $\mu := \mathsf{max\_improve}_{\mathcal{M}}(\mu, \boldsymbol{d})$
  $\boldsymbol{d} := \mathit{lfp}[\![\mu]\!]$
done
return $\boldsymbol{d}$

Fig. 2. Max-strategy iteration algorithm

The least fixed point $\mathit{lfp}[\![\mu]\!]$ of a strategy μ can be computed by solving the LP problem with the constraint system

$$\text{for each } \left(\boldsymbol{\delta}_{\ell'} = [\![R]\!]^\sharp(\boldsymbol{\delta}_\ell)\right) \text{ in } \mu : \boldsymbol{d}_{\ell'} \leq \mathbf{T}_{\ell'}\boldsymbol{x}' \ \wedge\ \mathbf{T}_\ell\boldsymbol{x} \leq \boldsymbol{d}_\ell \ \wedge\ R(\boldsymbol{x}, \boldsymbol{x}')$$

(where $\boldsymbol{x}$ and $\boldsymbol{x}'$ are auxiliary variables) and the objective function $\max \sum_i d_i$, *i.e.* the sum of all bounds $\boldsymbol{d}$.

μ' is called an *improvement* of μ w.r.t. $\boldsymbol{d}$, *i.e.* $\mu' = \mathsf{max\_improve}_{\mathcal{M}}(\mu, \boldsymbol{d})$ iff

1. μ' is "at least as good" as μ: $[\![\mu']\!](\boldsymbol{d}) \geq [\![\mu]\!](\boldsymbol{d})$, and
2. μ' is "strictly better for the changed equations": if $(\delta_i = \widehat{F}_i(\boldsymbol{\delta}))$ in μ and $(\delta_i \geq \widehat{F}'_i(\boldsymbol{\delta}))$ in μ' and $\widehat{F}_i \neq \widehat{F}'_i$, then $\widehat{F}'(\boldsymbol{d}) > \widehat{F}(\boldsymbol{d})$.

Example 4 (Max-strategy iteration). We illustrate some steps of the analysis of the example in Fig. 1. Assume the current strategy is:

$$\mu_1 = \left\{ \begin{array}{l} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = \sup\{\ \ x' \mid x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 5\ \ \} \\ \delta_{1,2} = \sup\{ -x' \mid x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 5\ \ \} \\ \delta_{2,1} = -\infty \\ \delta_{2,2} = -\infty \end{array} \right\}$$

and the current template bounds are:

$$\boldsymbol{d}_1 = \left\{ \begin{array}{ll} \delta_{0,1} \to \infty & \delta_{0,2} \to \infty \\ \delta_{1,1} \to 5 & \delta_{1,2} \to -5 \\ \delta_{2,1} \to -\infty & \delta_{2,2} \to -\infty \end{array} \right\}$$

The strategy can only be improved w.r.t. $\delta_{1,1}$:

$$\mu_2 = \left\{ \begin{array}{l} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = \sup\{\ \ x' \mid x \leq \delta_{1,1} \ \wedge\ -x \leq \delta_{1,2} \ \wedge\ x \leq 9 \ \wedge\ x' = x+1\ \} \\ \delta_{1,2} = \sup\{ -x' \mid x \leq \delta_{0,1} \wedge -x \leq \delta_{0,2} \wedge x' = 5\ \ \ \ \ \ \ \ \ \ \} \\ \delta_{2,1} = -\infty \\ \delta_{2,2} = -\infty \end{array} \right\}$$

We compute the new fixed point w.r.t. μ_2:

$$\boldsymbol{d}_2 = lfp[\![\mu_2]\!] = \left\{ \begin{array}{ll} \delta_{0,1} \to \infty & \delta_{0,2} \to \infty \\ \delta_{1,1} \to 10 & \delta_{1,2} \to -5 \\ \delta_{2,1} \to -\infty & \delta_{2,2} \to -\infty \end{array} \right\}$$

In the next step we can improve the strategy w.r.t. $\delta_{2,1}$ and $\delta_{2,2}$, a.s.o.

An improving strategy is selected by testing for each equation whether an argument of its max-operator leads to a greater bound. Since the arguments of the max-operator are required to be monotonic, the bounds are always monotonically increasing and, thus, arguments that have already been selected in previous strategies need not be considered again.

3 Logico-Numerical Max-Strategy Iteration

We will present an algorithm which enables the use of max-strategy iteration in a logico-numerical context, *i.e.* programs with a state space $\mathbb{B}^p \times \mathbb{R}^n$.

3.1 Abstract Domain

We consider the logico-numerical abstract domain $A = \wp(\mathbb{B}^p) \times \overline{\mathbb{R}}^m$ which combines Boolean formulas with template polyhedra. An abstract value $(B, \boldsymbol{d})$ consists of the cartesian product of valuations of the Boolean variables B (represented as Boolean formulas using BDDs for example) and the template bounds $\boldsymbol{d}$. We define the domain operations:

- Abstraction: $\alpha_{\mathbf{T}}(S) = \begin{pmatrix} \{\boldsymbol{b} \mid \exists \boldsymbol{x} : (\boldsymbol{b}, \boldsymbol{x}) \in S\} \\ \min\{\boldsymbol{d} \mid (\boldsymbol{b}, \gamma^x_{\mathbf{T}}(\boldsymbol{d})) \in S\} \end{pmatrix}$
- Concretization: $\gamma_{\mathbf{T}}(S^\sharp) = B \wedge \gamma^x_{\mathbf{T}}(\boldsymbol{d})$
- Join: $\begin{pmatrix} B \\ \boldsymbol{d} \end{pmatrix} \sqcup_{\mathbf{T}} \begin{pmatrix} B' \\ \boldsymbol{d}' \end{pmatrix} = \begin{pmatrix} B \vee B' \\ \boldsymbol{d} \sqcup^x_{\mathbf{T}} \boldsymbol{d}' \end{pmatrix}$
- Image: $[\![R_{\ell,\ell'}]\!]^\sharp \begin{pmatrix} B \\ \boldsymbol{d} \end{pmatrix} = \begin{pmatrix} \{\boldsymbol{b}' \mid \mathbf{T}_\ell \boldsymbol{x} \leq \boldsymbol{d} \wedge \boldsymbol{b} \in B \wedge R(\boldsymbol{b}, \boldsymbol{b}', \boldsymbol{x}, \boldsymbol{x}')\} \\ \sup\{\mathbf{T}_{\ell'} \boldsymbol{x}' \mid \mathbf{T}_\ell \boldsymbol{x} \leq \boldsymbol{d} \wedge \boldsymbol{b} \in B \wedge R(\boldsymbol{b}, \boldsymbol{b}', \boldsymbol{x}, \boldsymbol{x}')\} \end{pmatrix}$

$\top$ and $\bot$ are defined as $\begin{pmatrix} \mathsf{tt} \\ \infty \end{pmatrix}$ and $\begin{pmatrix} \mathsf{ff} \\ -\infty \end{pmatrix}$ respectively.

Since we are analyzing a CFG with locations L, we have the overall abstract domain $\Sigma^\sharp = L \to A$. An abstract value $S^\sharp = \lambda\ell.(B_\ell, \boldsymbol{d}_\ell) \in \Sigma^\sharp$ assigns to each location a value of the above logico-numerical domain. Note that the dimension m of $\boldsymbol{d}_\ell$ may depend on ℓ if the templates differ from location to location.

3.2 Algorithm

Our analysis is based on alternating (1) a truncated Kleene iteration over the logico-numerical abstract domain and (2) numerical max-strategy iteration, see Fig. 3.

```
1  S := S^0
2  S' = post(S)
3  while ¬stable(S, S') do
4     while ¬p_stable(S, S') do
5        S := S'
6        S' = post(S)
7     done
8     S := S'
9     M = generate(S)
10    μ := (δ = d)
11    μ' = max_improve_M(μ, d)
12    while μ' ≠ μ do
13       μ := μ'
14       d := lfp⟦μ⟧
15       μ' = max_improve_M(μ, d)
16    done
17    S' = post(S)
18 done
19 return S
```

lines 4–7: phase (1): truncated logico-numerical Kleene iteration

lines 12–16: phase (2): numerical max-strategy iteration

Fig. 3. Logico-numerical max-strategy iteration algorithm

The truncated Kleene iteration (propagation) explores the system until a certain criterion is satisfied; we say that the system *preliminarily stable*. We use the following criterion: we stop Kleene iteration if for all locations the set of reachable Boolean states does not change whatever transition we take. The underlying idea is to discover a subsystem, in which Boolean variables are stable during a presumably larger number of iterations. In such a subsystem numerical variables evolve, while Boolean transitions switch only within the system, *i.e.* they do not "discover" new Boolean states, until numerical conditions are satisfied that make Boolean variables leave the subsystem.

We use max-strategy iteration (phase (2)) to compute the fixed point for the numerical variables for such a subsystem. Then Kleene iteration (phase (1)) continues exploring the next preliminary stable subsystem. The algorithm terminates in a finite number of steps as soon as the Kleene iteration of phase (1) has reached a fixed point.

Formal Description. See Fig. 3. Since we only manipulate abstract quantities, we will omit the superscript $^\sharp$ in the sequel in order to improve readability.

For phase (1) we use the definitions:

- Initial abstract value: $S^0 = \lambda\ell. \begin{cases} \top \text{ for } \ell = \ell_0 \\ \bot \text{ for } \ell \neq \ell_0 \end{cases}$
- Post-condition: $\mathsf{post}(S) = \lambda\ell'.S(\ell') \sqcup \bigsqcup_\ell [\![R_{\ell,\ell'}]\!](S(\ell))$
- Condition for preliminary stability: $\mathsf{p\_stable}(S, S') = (\forall \ell : B_\ell = B'_\ell)$

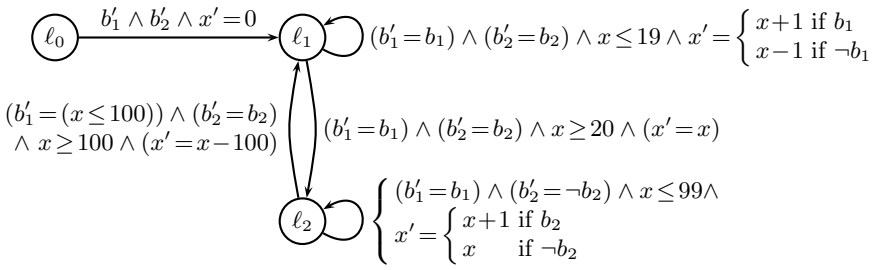


Fig. 4. CFG of the program in Example 1

– Condition for stability (global convergence): $\mathsf{stable}(S, S') = (S = S')$

For phase (2) we define the following:

– The max-strategy improvement operator $\mathsf{max\_improve}_{\mathcal{M}}$ is defined as described in §2.
– The operator $\mathsf{generate}$ dynamically derives the system of equations for the current preliminary stable subsystem: this means that we restrict the system to those transitions which stay within the subsystem defined by the current Boolean states $\lambda\ell.B_\ell$. For this purpose we conjoin the term $\boldsymbol{b} \in B_\ell \wedge \boldsymbol{b'} \in B_{\ell'}$ to the transition relation in the definition below. Strategies may only contain convex constraints: thus, we transform the relation into disjunctive normal form and generate one strategy per disjunct:

$$\mathsf{generate}(S) = \bigcup_{\ell',\ell} \mathsf{decomp\_convex}(\exists \boldsymbol{b}, \boldsymbol{b'} : R_{\ell,\ell'}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{b'}, \boldsymbol{x'}) \wedge \boldsymbol{b} \in B_\ell \wedge \boldsymbol{b'} \in B_{\ell'})$$

where $\mathsf{decomp\_convex}(R_{\ell,\ell'}) = \bigcup_j \left(\boldsymbol{\delta}_{\ell'} = [\![R^j_{\ell,\ell'}]\!](\boldsymbol{\delta}_\ell)\right)$ with $R_{\ell,\ell'} = \bigvee_j R^j_{\ell,\ell'}$ and $R^j_{\ell,\ell'}$ convex.

Remark 1. Since the bounds $\boldsymbol{d}$ are monotonically increasing, we use the constant strategy $\boldsymbol{\delta} = \boldsymbol{d}$ (where $\boldsymbol{d}$ are the previously obtained bounds) as initial strategy for phase (2) (see line 10 in Fig. 3). This prevents the numerical max-strategy improvement from restarting with $\boldsymbol{\delta} = -\boldsymbol{\infty}$ each time.

We illustrate this algorithm by applying it to the CFG in Fig. 4:

Example 5. We use the template constraint matrix $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ which corresponds to an interval analysis. In order to make the presentation more concise, we will write states $(\ell \to \begin{pmatrix} B \\ \boldsymbol{d} \end{pmatrix}) \in \Sigma$ as $\ell \to \begin{pmatrix} \varphi(b_1, b_2) \\ [-\delta_{\ell,2}, \delta_{\ell,1}] \end{pmatrix}$ where φ is a Boolean formula.

The initial state in ℓ_0 is $\begin{pmatrix} \mathsf{tt} \\ [-\infty, \infty] \end{pmatrix}$. We start exploring the system by taking transition (ℓ_0, R, ℓ_1): $\ell_1 \to \begin{pmatrix} b_1 \wedge b_2 \\ [0, 0] \end{pmatrix}$. We continue propagating through

(ℓ_1, R, ℓ_1): $\ell_1 \to \begin{pmatrix} b_1 \wedge b_2 \\ [0,1] \end{pmatrix}$. Now, we have reached preliminary stability because none of the transitions makes us discover new Boolean states in the next iteration ((ℓ_0, R, ℓ_1) and (ℓ_1, R, ℓ_1) yield nothing new w.r.t. Boolean states and the other transitions are infeasible, *i.e.*, they give $\bot$). Hence, we go ahead to phase (2) and extract the numerical equation system for each (ℓ, R, ℓ'). *E.g.* for (ℓ_1, R, ℓ_1), we compute:

$$\exists \boldsymbol{b}, \boldsymbol{b'} : R \wedge b_1 \wedge b_2 \wedge b'_1 \wedge b'_2 = (x' = x+1 \wedge x \leq 19)$$

which gives us the partial equations:

$$\begin{array}{l} \delta_{1,1} = \sup\{ \quad x' \mid x \leq \delta_{1,1} \wedge -x \leq \delta_{1,2} \wedge x' = x+1 \wedge x \leq 19 \ \} \\ \delta_{1,2} = \sup\{ -x' \mid x \leq \delta_{1,1} \wedge -x \leq \delta_{1,2} \wedge x' = x+1 \wedge x \leq 19 \ \} \end{array}$$

which have to be completed by the other incoming arcs of ℓ_1. We start the max-strategy iteration with the strategy corresponding to the values obtained in phase (1):

$$\mu = \{\boldsymbol{\delta}_0 = \boldsymbol{\infty}, \qquad \delta_{1,1} = 1, \qquad \delta_{1,2} = 0, \qquad \boldsymbol{\delta}_2 = \boldsymbol{-\infty}\}$$

We observe that we can improve this strategy w.r.t. $\delta_{1,1}$:

$$\mu' = \left\{ \begin{array}{l} \boldsymbol{\delta}_1 = \boldsymbol{\infty} \\ \delta_{1,1} = \sup\{x' \mid x \leq \delta_{1,1} \ \wedge \ -x \leq \delta_{1,2} \ \wedge \ x' = x+1 \ \wedge \ x \leq 19\}, \quad \delta_{1,2} = 0 \\ \boldsymbol{\delta}_2 = \boldsymbol{-\infty} \end{array} \right\}$$

The max-strategy iteration terminates with: $\ell_1 \to \begin{pmatrix} b_1 \wedge b_2 \\ [0,20] \end{pmatrix}$.

We continue propagating (phase (1)): By (ℓ_1, R, ℓ_2) we get $\ell_2 \to \begin{pmatrix} b_1 \wedge b_2 \\ [20,20] \end{pmatrix}$; then (ℓ_2, R, ℓ_2) results in $\begin{pmatrix} b_1 \wedge \neg b_2 \\ [21,21] \end{pmatrix}$; by joining these values we get $\ell_2 \to \begin{pmatrix} b_1 \\ [20,21] \end{pmatrix}$. Taking (ℓ_2, R, ℓ_2) a second time does not change the Boolean state: $\ell_2 \to \begin{pmatrix} b_1 \\ [20,22] \end{pmatrix}$. Taking any other transition does not discover new Boolean states either, thus, we move on to phase (2) and compute the numerical equation system w.r.t. the current Boolean state: For example for (ℓ_2, R, ℓ_2), we compute

$$(\exists \boldsymbol{b}, \boldsymbol{b'} : R \wedge b_1 \wedge b'_1) = \big((x' = x+1 \vee x' = x) \wedge x \leq 99\big)$$

which results in the partial equations

$$\begin{array}{l} \delta_{2,1} = \max \left\{ \begin{array}{l} \sup\{ \ x' \mid x \leq \delta_{2,1} \wedge -x \leq \delta_{2,2} \wedge x' = x+1 \wedge x \leq 99 \ \}, \\ \sup\{ \ x' \mid x \leq \delta_{2,1} \wedge -x \leq \delta_{2,2} \wedge x' = x \quad \wedge x \leq 99 \ \} \end{array} \right\} \\ \delta_{2,2} = \max \left\{ \begin{array}{l} \sup\{ -x' \mid x \leq \delta_{2,1} \wedge -x \leq \delta_{2,2} \wedge x' = x+1 \wedge x \leq 99 \ \}, \\ \sup\{ -x' \mid x \leq \delta_{2,1} \wedge -x \leq \delta_{2,2} \wedge x' = x \quad \wedge x \leq 99 \ \} \end{array} \right\} \end{array}$$

which have to be completed by the other incoming arcs of ℓ_2. The only possible improvement w.r.t to the current state is w.r.t. $\delta_{2,1}$; phase (2) results in $\ell_2 \to \begin{pmatrix} b_1 \\ [20,100] \end{pmatrix}$.

We continue with phase (1), which filters the above value through (ℓ_2, R, ℓ_1) augmenting the abstract value in ℓ_1 to $\begin{pmatrix} b_1 \\ [0,20] \end{pmatrix}$. Then, *none* of the transitions makes the reachable state sets increase (neither Boolean nor numerical), hence we have reached the global fixed point:

$$\ell_0 \to \top, \quad \ell_1 \to \begin{pmatrix} b_1 \\ [0,20] \end{pmatrix}, \quad \ell_2 \to \begin{pmatrix} b_1 \\ [20,100] \end{pmatrix}$$

Note that a logico-numerical analysis in the same domain with widening and descending iterations yields no information about this example: $S = \lambda\ell.\top$.

3.3 Properties

Theorem 1 (Termination). *The logico-numerical max-strategy algorithm terminates after a finite number of iterations.*

Proof. Termination follows from these observations:
(a) Phase (1) only propagates as long as new Boolean states are discovered; the number of Boolean states is finite.
(b) Max-strategy iteration is called at most once for each subset of Boolean states. The number of subsets of Boolean states is finite.
(c) There is a unique system of numerical equations (built by generate) for each subset of Boolean states.
(d) Max-strategy iteration terminates after a finite number of improvement steps, because there is a finite number of strategies and each strategy is visited at most once [8].
(e) Max-strategy iteration returns the unique least fixed point w.r.t. the given system of equations [8].
Thus, as soon as all reachable Boolean states have been discovered and the associated numerical fixed point has been computed, the overall fixed point has been reached and the algorithm terminates.

Theorem 2 (Soundness). *The logico-numerical max-strategy algorithm computes a fixed point of* $\lambda S.S_0 \sqcup [\![R]\!]^\sharp(S)$.

Proof. Let us denote
- $F = \lambda S.S_0 \sqcup [\![R]\!]^\sharp(S)$.
- $\lambda S.(lfp^B\ F)(S)$ the truncated Kleene iteration phase (1) (lines 4 to 7 in Fig. 3), *i.e.* $\lambda S.\big(\text{while } B \neq B' \text{ do } S := S'; S' = F(S) \text{ end; return } S'\big)$.

- $\lambda(B, \boldsymbol{d}).(B, (lfp^X\ F^X)(\boldsymbol{d}))$ the max-strategy iteration in phase (2) (lines 9 to 16 in Fig. 3),
  i.e. $\lambda S.\big(\ \mathcal{M} = \mathsf{generate}(S); \mu := (\boldsymbol{\delta} = \boldsymbol{d}); \mu' = \mathsf{max\_improve}_{\mathcal{M}}(\mu, \boldsymbol{d});$ while $\mu' \neq \mu$ do $\mu := \mu'; \boldsymbol{d} := lfp[\![\mu]\!]; \mu' = \mathsf{max\_improve}_{\mathcal{M}}(\mu, \boldsymbol{d})$ done; return $S\big)$.

Then, we can write the whole algorithm as $lfp((id^B, lfp^X\ F^X) \circ (lfp^B\ F) \circ F)$.

Since $lfp^B\ F$ and $(id^B, lfp^X\ F^X)$ are both extensive, we can under-approximate them by $id = \lambda S.S$. Hence, we conclude from

$$lfp\ F \sqsubseteq lfp(\underbrace{id}_{id \sqsubseteq (id^B, lfp^X F^X)} \circ \underbrace{id}_{id \sqsubseteq (lfp^B F)} \circ F)$$

that our algorithm computes an over-approximation of the least fixed point, *i.e.* it is sound.

Theorem 3 (Optimality). *The logico-numerical max-strategy algorithm computes the* least *fixed point of* $\lambda S.S_0 \sqcup [\![R]\!]^\sharp(S)$.

Proof. Additionally to Thm. 2, we have to show that

$$lfp((id^B, lfp^X\ F^X) \circ (lfp^B\ F) \circ F) \sqsubseteq lfp\ F.$$

(a) Phase (1) computes a certain number of iterations $F^k(\bot) = (lfp^B\ F) \circ F(\bot)$ taking into account the whole transition system. We trivially have $F^k(\bot) \sqsubseteq lfp\ F(\bot)$.

(b) Phase (2) $(id^B, lfp^X\ F^X)$ iterates over the transitions of a subsystem. It is known [8] that it computes the least fixed point w.r.t. this subsystem. Hence, the result of phase (2) cannot go beyond the fixed point of the whole system: $(id^B, lfp^X\ F^X) \circ F^k(\bot) \sqsubseteq lfp\ F$.

(c) We can repeat arguments (a) and (b) for the outer loop:
$\ldots \circ (id^B, lfp^X\ F^X) \circ F^{k_2} \circ (id^B, lfp^X\ F^X) \circ F^{k_1}(\bot) \sqsubseteq lfp\ F$
where k_n is the number of iterations of the n^{th} phase (1).
Thus, we have $((id^B, lfp^X\ F^X) \circ F^{k_n})^n(\bot) \sqsubseteq lfp\ F$ for $n \geq 0$.

Hence, we conclude from $lfp((id^B, lfp^X\ F^X) \circ (lfp^B\ F) \circ F) \sqsubseteq lfp\ F$ and Thm. 2 that our algorithm computes the least fixed point, *i.e.* it is optimal.

3.4 Application to Data-Flow Programs

For our experiments in §4, we used Lustre programs, *i.e.* synchronous data-flow programs. For this reason we will briefly explain how to apply our algorithm to such programs.

Lustre programs can be reduced to a symbolic transition system $\begin{cases} \mathcal{I}(\boldsymbol{s}) \\ \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i}) \end{cases}$ where $\boldsymbol{s} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{x} \end{pmatrix}$ and $\boldsymbol{i} = \begin{pmatrix} \boldsymbol{\beta} \\ \boldsymbol{\xi} \end{pmatrix}$ are the vectors of (Boolean and numerical) state and input variables, $\mathcal{I}(\boldsymbol{s})$ is the initial condition, $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$ is an *assertion* constraining input variables depending on state variables, and $\boldsymbol{f}$ is the vector of transition functions.

State space partitioning is used to obtain a CFG in which each equivalence class of the partition corresponds to a location.

The transition relations are constructed by

$$R_{\ell,\ell'} = \exists \boldsymbol{\beta} : \left\{ \begin{array}{l} \boldsymbol{x}' = \boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{\xi}) \\ \boldsymbol{b}' = \boldsymbol{f}^b(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{\xi}) \end{array} \right\} \wedge \psi_\ell(\boldsymbol{x}, \boldsymbol{b}) \wedge \psi_{\ell'}(\boldsymbol{x}', \boldsymbol{b}') \wedge \mathcal{A}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{\xi})$$

where $f_i^x = \left\{ \begin{array}{l} \cdots \\ a_{ij}(\boldsymbol{x}, \boldsymbol{\xi}) \text{ if } \phi_{ij}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{\xi}) \\ \cdots \end{array} \right.$, and ψ_ℓ are the definitions of the partitions (locations).

Boolean input variables are quantified existentially. Numerical inputs appear as auxiliary variables (*i.e.*, variables without associated bounds) in the max-strategy iteration, hence, they are treated without modification of the algorithms.

3.5 Discussion

An important observation is that, since the overall abstract domain is of the form $L \to \wp(\mathbb{B}^p) \times \overline{\mathbb{R}}^m$, the choice of the CFG has two effects on the performance: first, it determines the set of representable abstract properties, and second, it influences the approximations made in the generation of the numerical equation system for the max-strategy iteration phase (2), because there is only one template polyhedron per location.

Generalization. The structure of the algorithm we presented is quite general. In particular, it does not depend on the method used to compute the numerical least fixed point in phase (2). We conjecture that the algorithm makes every method, that is able to compute the least fixed point of a numerical system by ascending iterations, compute the least fixed point of a logico-numerical system.

For example, we suppose that our algorithm can be used without any modification with the variant of max-strategy iteration for quadratic programs and quadratic templates proposed in [19].

If a method computes the fixed point by descending iterations, as for example min-strategy iteration [4, 5], our algorithm can still be used, but requires a small adaptation because the abstract value computed in phase (1), which is an under-approximation of the least fixed point, cannot be used to initialize phase (2), which requires an over-approximation: hence, line 10 in Fig. 3 must be replaced by guessing appropriate initial bounds and an initial strategy for phase (2). This makes the algorithm less elegant and the analysis, probably, less efficient.

Logico-Numerical Max-Strategy Iteration Using a Power Domain. The algorithm is also rather generic w.r.t. the kind of logico-numerical abstract domain we use. For example, we could consider the logico-numerical power domain $\mathbb{B}^p \to \wp(\mathbb{R}^n)$ where $\wp(\mathbb{R}^n)$ is abstracted by any domain that is supported by strategy iteration. Then the overall domain for our method is $L \to \mathbb{B}^p \to \overline{\mathbb{R}}^m$. This domain implicitly dynamically partitions each location into sub-locations corresponding to Boolean valuations sharing a common numerical abstract value[3]. The construction of the equation system (`generate` in our algorithm, Fig. 3) must take into account these partitions.

[3] This sharing can be implemented with the help of MTBDDs where the numerical abstract values are stored in the leaves [21].

This domain is more precise than the product domain described in §3, however, the drawback is that the number of partitions might explode if only few Boolean valuations share a common numerical abstract value.

Comparison with Logico-Numerical Min-Strategy Iteration. The power domain $\mathbb{B}^p \to \overline{\mathbb{R}}^m$ is also used by Sotin et al [22] who propose an approach to analyzing logico-numerical programs using min-strategies. In accordance with the form of the abstract domain, they consider logico-numerical strategies $\mathbb{B}^p \to \Pi$ (where Π is the set of min-strategies), which dynamically associates the numerical min-strategies to the reachable Boolean states during analysis. They start with an initial logico-numerical strategy $P^{(0)} = \lambda \boldsymbol{b}.\pi^{(0)}$ with a chosen numerical min-strategy $\pi^{(0)}$ and compute a fixed point using logico-numerical Kleene iteration with widening and descending iterations. Then they iteratively improve the min-strategies in $P^{(i)}$ and recompute the fixed point.

This approach does not integrate well with mathematical programming because the only known method for computing the fixed point of a logico-numerical strategy is logico-numerical Kleene iteration (with widening). Hence, in contrast to our approach, there is no guarantee to compute the least fixed point.

Comparison with Abstract Acceleration. Numerous methods have been developed to alleviate the problem of bad extrapolations due to widening, *e.g.* abstract acceleration [23–25], a method for computing the transitive closure of numerical loops. These methods are able to accelerate some cases of self-loops and cycles with certain types of affine transformations, and they rely on widening in the general case. However, due to the use of general convex polyhedra, they are able to "discover" complex invariant constraints.

In contrast, max-strategy iteration is able to "accelerate" globally the whole transition system regardless of the graph structure or type of affine transformation, and it effectively computes the least fixed point. However, this is only possible on the simpler domain of template polyhedra.

Although the use of template polyhedra is a restriction, this kind of (static) approximation is much more predictable than the (dynamic) approximations made by widening.

Remark 2. Guided static analysis [26] is a framework for analyzing monotonically increasing subsystems, which makes it possible to reduce the impact of widening by applying descending iterations "in the middle" of an analysis. Our algorithm proceeds in a similar fashion – although for different technical reasons – by applying max-strategy iteration on monotonically increasing subsystems.

4 Experimental Evaluation

We implemented the algorithm in our reactive system verification tool ReaVer[4], which is based on the logico-numerical abstract domain library BddApron [21] and an improved version of the max-strategy iteration solver of Gawlitza et al

[4] http://pop-art.inrialpes.fr/people/schramme/reaver/

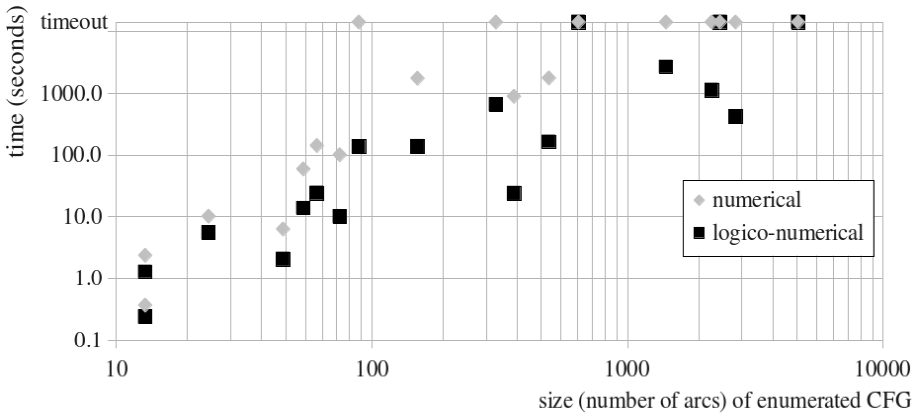


Fig. 5. Scalability of *logico-numerical* max-strategy iteration in comparison with *numerical* max-strategy iteration on the enumerated CFG, using octagonal constraints. The timeout was set to 3600 seconds. Note the logarithmic scales.

[27]. Since template polyhedra are not yet implemented in the APRON library [28], we emulated template polyhedra operations in phase (1) with the help of general polyhedra, which certainly impaired the performance – nonetheless we obtained encouraging results.

Benchmarks. We took 18 benchmarks[5] used in [25]. These are high-level simulation models (programmed in LUSTRE) of manufacturing systems which consist of building blocks like sources, buffers, machines and routers that synchronize via handshakes (for this reason there are many Boolean variables). The properties to be verified like maximal throughput time depend on numerical variables. These programs have up to a few hundred lines of code, 27 Boolean and 7 numerical variables, which produce enumerated CFGs of up to 650 locations and 5000 transitions after simplification by Boolean reachability. The focus of the experiments was on comparing the precision of the inferred invariants rather than proving properties.

Results. We performed experiments with octagonal constraints ($\pm x_i$, $x_i \pm x_j$) in order to evaluate efficiency and precision. We compared max-strategy iteration on the enumerated CFGs (MSI) with logico-numerical max-strategy iteration (LNMSI) on CFGs obtained by the static partitioning method by Boolean states implying the same numerical transitions ("numerical modes") from [25]. The resulting CFGs are on average five times smaller than the enumerated CFGs for the medium-sized benchmarks.

– LNMSI scales clearly better than MSI (see Fig. 5): our method was on average 9 times faster – for those benchmarks where both methods terminated before

[5] The benchmarks can be downloaded from:
http://pop-art.inrialpes.fr/people/schramme/maxstrat/

the timeout: MSI hit the timeout in 8 out of 18 cases (versus 3 for our method). The gain in efficiency grows with increasing benchmark sizes.
– The precision is almost preserved: only 0.38% (!) of the bounds were worse but still finite, and 0.16% were lost. This precision loss did not impact the number of proved properties. Due to the better scalability we were even able to prove 3 more benchmarks (10 as opposed to 7).
– We compared the precision of LNMSI with octagonal constraints with a logico-numerical analysis with octagons using the standard approach with widening ($N=2$) and 2 descending iterations on the same CFG. On average 18% of the bounds of our invariants were strictly better than those computed using the standard analysis. In 2 cases these improvements made the difference to prove the property. However, the standard analysis was 19 times faster on average.

Furthermore, we experimented with different templates and CFG sizes:

– The average gain in speed increases with the template size: 3.3 for interval analysis ($\pm x_i$), 5 for zones ($\pm x_i$, $x_i - x_j$) and 9 for octagons (for those benchmarks which did not run into timeouts).
– The precision of LNMSI depends on the CFG size: the general trend is "the bigger the more precise", but the results are less clear: CFGs of the same size seem to have very different *quality* w.r.t. precision. Partitioning methods which find *good* partitions matter!
– A smaller CFG does not automatically mean faster analysis: the fact that a smaller graph means more complicated logico-numerical transition functions and more numerical strategies per location outweighs the advantage of dealing with less locations.
– It is interesting that LNMSI scales also better on the enumerated CFG: it seems to be advantageous to start with a small system with few strategies, iteratively increase the system, and finally, when computing the numerical fixed point of the full system, most of the strategies are already known not to improve the bounds, and thus max-strategy iteration converges faster.

We also experimented with LNMSI using the logico-numerical power domain discussed in §3.5, which performed on our CFGs still 6 to 7 times faster on average and with a 100% preservation of bounds compared to MSI.

5 Conclusions

We presented *logico-numerical max-strategy iteration*, a solution to the intricate problem of combining numerical max-strategy iteration with techniques that are able to deal with Boolean variables implicitly and therefore allow to trade precision for efficiency.

In contrast to the previous attempt of Sotin et al [22] of extending strategy iteration to logico-numerical programs, which relies on widening operators to converge, our method enables the use of mathematical programming and hence, it indeed computes the *best logico-numerical invariant* w.r.t. the chosen abstract domain.

The effectiveness of our method depends on two factors:

(1) The choice of the templates: in our experiments we used mainly octagonal constraints, but we could have used methods (*e.g.* [10]) for inferring template constraints.
(2) The considered CFG (either of the imperative program, or the one obtained by partitioning in the case of data-flow programs) which determines the abstract domain: the partitioning method by "numerical modes" turned out to be surprisingly effective: compared to the solution based on an enumeration of the reachable Boolean states, the obtained CFGs were 5 times smaller on average, still the precision loss was negligible, *i.e.* almost zero, and we gained at least one order of magnitude w.r.t. efficiency.

Furthermore, this paper delivers the first experimental results of applying *numerical* max-strategy iteration to larger programs: on the one hand max-strategy iteration is guaranteed to compute more precise invariants than standard techniques in the same domain, on the other hand our implementation is not (yet) able to compete with standard techniques w.r.t. efficiency.

Perspectives. Our algorithm is quite generic w.r.t. the numerical analysis method and logico-numerical abstract domain. Though, in order to tackle efficiency issues evoked above, it would be interesting to design a more integrated logico-numerical max-strategy solver. This would enable to share more information between subsequent calls to the max-strategy iteration, *e.g.* to avoid retesting of strategies that will definitely not lead to an improvement. Beyond that, we could more extensively use SMT-solvers. For instance, checking whether a strategy is an improvement is currently done after having constructed the numerical equation system; it would be beneficial to find the improving strategies already on the logico-numerical level.

Moreover, we plan to apply our method to the analysis of logico-numerical hybrid automata [29] by extending the hybrid max-strategy iteration method of Dang and Gawlitza [27, 30].

Acknowledgements. We thank Thomas M. Gawlitza, Bertrand Jeannet, Philipp Rümmer and the anonymous reviewers for their valuable remarks on the draft of this paper.

References

1. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: Principles of Programming Languages, pp. 178–188. ACM (1987)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252 (1977)
3. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages, pp. 84–97. ACM (1978)

4. Costan, A., Gaubert, S., Goubault, É., Martel, M., Putot, S.: A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 462–475. Springer, Heidelberg (2005)
5. Gaubert, S., Goubault, É., Taly, A., Zennou, S.: Static Analysis by Policy Iteration on Relational Domains. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 237–252. Springer, Heidelberg (2007)
6. Adjé, A., Gaubert, S., Goubault, E.: Coupling Policy Iteration with Semi-definite Relaxation to Compute Accurate Numerical Invariants in Static Analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 23–42. Springer, Heidelberg (2010)
7. Gawlitza, T., Seidl, H.: Precise Fixpoint Computation Through Strategy Iteration. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 300–315. Springer, Heidelberg (2007)
8. Gawlitza, T., Seidl, H.: Precise Relational Invariants Through Strategy Iteration. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 23–40. Springer, Heidelberg (2007)
9. Gawlitza, T., Seidl, H., Adjé, A., Gaubert, S., Goubault, E.: Abstract interpretation meets convex optimization. Journal of Symbolic Computation 47, 1416–1446 (2012)
10. 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)
11. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Dunod, pp. 106–130 (1976)
12. 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)
13. Miné, A.: The octagon abstract domain. In: Working Conference on Reverse Engineering, pp. 310–319. IEEE (2001)
14. Bultan, T., Gerber, R., Pugh, W.: Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 400–411. Springer, Heidelberg (1997)
15. Jeannet, B., Halbwachs, N., Raymond, P.: Dynamic Partitioning in Analyses of Numerical Properties. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 39–50. Springer, Heidelberg (1999)
16. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation, pp. 196–207. ACM (2003)
17. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: Formal Methods in Computer-Aided Design, pp. 1–9. IEEE (2008)
18. Gawlitza, T., Seidl, H.: Precise Interval Analysis vs. Parity Games. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 342–357. Springer, Heidelberg (2008)
19. Gawlitza, T.M., Seidl, H.: Computing Relaxed Abstract Semantics w.r.t. Quadratic Zones Precisely. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 271–286. Springer, Heidelberg (2010)
20. Gawlitza, T.M., Seidl, H.: Solving systems of rational equations through strategy iteration. Transactions on Programming Languages and Systems 33, 11 (2011)

21. Jeannet, B.: BDDAPRON: A logico-numerical abstract domain library (2009), http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/
22. Sotin, P., Jeannet, B., Védrine, F., Goubault, E.: Policy Iteration within Logico-Numerical Abstract Domains. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 290–305. Springer, Heidelberg (2011)
23. Gonnord, L., Halbwachs, N.: Combining Widening and Acceleration in Linear Relation Analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
24. Schrammel, P., Jeannet, B.: Applying abstract acceleration to (co-)reachability analysis of reactive programs. Journal of Symbolic Computation 47, 1512–1532 (2012)
25. Schrammel, P., Jeannet, B.: Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 233–248. Springer, Heidelberg (2011)
26. Gopan, D., Reps, T.W.: Guided Static Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer, Heidelberg (2007)
27. Dang, T., Gawlitza, T.M.: Discretizing Affine Hybrid Automata with Uncertainty. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 473–481. Springer, Heidelberg (2011)
28. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
29. Schrammel, P., Jeannet, B.: From hybrid data-flow languages to hybrid automata: A complete translation. In: Hybrid Systems: Computation and Control, pp. 167–176. ACM (2012)
30. Dang, T., Gawlitza, T.M.: Template-Based Unbounded Time Verification of Affine Hybrid Automata. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 34–49. Springer, Heidelberg (2011)

A Constraint Solver Based on Abstract Domains

Marie Pelleau[1], Antoine Miné[2], Charlotte Truchet[1], and Frédéric Benhamou[1]

[1] Université de Nantes, 2 rue de la Houssinière, Nantes
{firstName.lastName}@univ-nantes.fr
[2] École normale supérieure
45, rue d'Ulm, Paris
Antoine.Mine@ens.fr

Abstract. In this article, we apply techniques from Abstract Interpretation (a general theory of semantic abstractions) to Constraint Programming (which aims at solving hard combinatorial problems with a generic framework based on first-order logics). We highlight some links and differences between these fields: both compute fixpoints by iteration but employ different extrapolation and refinement strategies; moreover, consistencies in Constraint Programming can be mapped to non-relational abstract domains. We then use these correspondences to build an abstract constraint solver that leverages abstract interpretation techniques (such as relational domains) to go beyond classic solvers. We present encouraging experimental results obtained with our prototype implementation.

1 Introduction

Abstract Interpretation is a method to design approximate semantics of programs and provide sound answers to questions about their run-time behaviors [8,7]. Constraint Programming aims at solving, with reusable techniques, hard combinatorial problems expressed declaratively. This article studies the application of Abstract Interpretation techniques to Constraint Programming.

State of the Art. First introduced by Montanari [19], Constraint Programming (CP) relies on the idea that many problems can be expressed as conjunctions of first-order logic formulas, called constraints, each one representing a specific combinatorial feature of the problem [21]. Each constraint comes with *ad hoc* operators exploiting its internal structure to reduce the combinatorics. The constraints are then combined into generic solving algorithms. Much of the research effort in CP is focused on defining and improving constraints[1] and fine-tuning solving algorithms. CP now offers powerful techniques for combinatorial optimization, with many practical applications to scheduling, packing, layout design, frequency allocation, etc. Yet, solvers suffer from limitations. They are limited to non-relational domains, such as boxes or Cartesian products of integer sets. Moreover, two clearly separate family of solving algorithms exist: one handles

[1] See http://www.emn.fr/z-info/sdemasse/gccat for a catalog of existing global constraints.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 434–454, 2013.

discrete variables and the other continuous variables. Several methods have been proposed to handle mixed problems, such as discretizing continuous variables and handling them in a discrete solver (as in Choco [23]). Unfortunately, the solver remains a purely discrete one and does not benefit from heuristics developed for continuous ones. Alternatively, one can add specific mixed constraints [6] or generic integrity constraints [4] to a continuous solver, with similar drawbacks.

In another research area, Abstract Interpretation (AI) is used to design static program analyzers that are sound and always terminate (such as Astrée [5]) by developing computable approximations of essentially undecidable problems. The (uncomputable) concrete collecting semantics expresses in fixpoint form the set of observable behaviors of the program. It is approximated in an abstract domain that restricts the expressiveness to a set of properties of interest, provides data-structure representations, efficient algorithms to compute abstract versions of concrete operators, and acceleration operators to approximate fixpoints in finite time. Soundness guarantees that the analyzer observes a super-set of the program behaviors. Numeric domains, focusing on numeric variables and properties, are particularly well developed; major ones include intervals [7] and polyhedra [9], and recent years have seen the development of new domains, such as octagons [18], and libraries, such as Apron [15]. They can handle all kinds of numeric variables, including mathematical integers, rationals, and reals, machine integers and floating-point numbers, and even express relationships between variables of different types [17,5]. Each domain corresponds to some trade-off between cost and precision. Finally, domains can be modified and combined by generic operators, such as disjunctive completions and reduced products.

Contribution. In this paper we seek to use AI techniques to build an abstract, generic CP solver. Our contributions are as follows: we show the links between AI and CP and recast the later as a fixpoint computation similar to local iterations in a disjunctive completion of non-relational domains; we design a generic abstract solver parametrized by abstract domains and prove its termination; we show that, by using relational and mixed integer-real abstract domains, we can go beyond some limitations of existing solvers. We do not study in this paper the dual problem, *i.e.,* exploiting CP techniques in AI; it is one of the perspectives of this work.

The paper is organized as follows. Section 2 provides background information on AI and CP, and some elements of comparison. Section 3 recasts CP as AI and presents our abstract solver. Our prototype implementation and preliminary experimental results are presented in Sec. 4. Section 5 concludes.

Related Works. Some interactions between CP and verification techniques have been explored in previous works. For instance, CP has been used to automatically generate test configurations [13], or to verify CP models [16]. In another direction, several recent works, such as [10,24], establish connections between AI and SAT solving algorithms, holding promise for cross-pollination between these fields. Our aim is similar, but linking AI to CP. While related, CP and SAT differ significantly enough in the chosen models (numeric versus

boolean) and solving algorithms that previous results do not apply. Our work is in the continuity of [25] that extends CP solving methods to use richer domain representations, such as octagons. However, embedding a new domain required *ad hoc* techniques to express its operations in the native language of the solver: boxes. In this paper, we reverse that process: we redesign from the ground up the solver in an abstract way so that it is not tied to boxes but can reuse as-is existing abstract operators and domains from AI.

2 Preliminaries

In this section we present some notions of Abstract Interpretation and Constraint Programming that will be needed later.

2.1 Bases of Abstract Interpretation

We first present some elements of Abstract Interpretation that will prove useful in the design of our solver (see [8,7] for a more detailed presentation).

Fix-Point Abstractions. The concrete semantics of a program is given as the least fixpoint $\text{lfp}_\bot F$ of an operator $F : \mathcal{D} \to \mathcal{D}$ in some partially ordered structure $(\mathcal{D}, \sqsubseteq, \bot, \sqcup)$, such as a complete partial order or a lattice. With suitable hypotheses [8] on F and $\mathcal{D}$, the fixpoint can be expressed as the limit of a (possibly transfinite) increasing iteration $\text{lfp}_\bot F = \bigsqcup_{i \in \text{Ord}} F^i(\bot)$ on ordinals.

Similarly, we denote by $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$ the abstract domain. A monotonic concretization $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ associates a concrete meaning to each abstract element. An abstract operator $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a sound abstraction of F if $F \circ \gamma \sqsubseteq \gamma \circ F^\sharp$. Sometimes, but not always, there exists an abstraction function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ such that (α, γ) forms a Galois connection, which ensures that each concrete element X has a best abstraction $\alpha(X)$, and the optimal abstract operator $F^\sharp$ can be uniquely defined as $F^\sharp = \alpha \circ F \circ \gamma$. In all cases, $\text{lfp}_\bot F$ can be approximated as $\bigsqcup^\sharp_{i \in \text{Ord}} F^{\sharp i}(\bot^\sharp)$. This limit may not be computable, even if $F^\sharp$ is, or may require many iterations. It is thus often replaced with the limit of an increasing sequence: $X^\sharp_0 = \bot^\sharp$, $X^\sharp_{i+1} = X^\sharp_i \triangledown F^\sharp(X^\sharp_i)$ using a widening operator $\triangledown$ to accelerate convergence. The widening is designed to over-approximate $\sqcup$ and converge in finite time δ to a post-fixpoint $X^\sharp_\delta$ of $F^\sharp$. Then, $\gamma(X^\sharp_\delta) \sqsupseteq \text{lfp}_\bot F$. The limit is often refined by a decreasing iteration: $Y^\sharp_0 = X^\sharp_\delta$, $Y^\sharp_{i+1} = Y^\sharp_i \vartriangle F^\sharp(Y^\sharp_i)$, using a narrowing operator $\vartriangle$ designed to stay above any fixpoint of F and converge in finite time. As all the $Y^\sharp_i$ are abstractions of $\text{lfp}_\bot F$, we can stop the iteration at any time.

Local Iterations. In addition to refining the results of least fixpoint computations, decreasing iterations have been used by Granger [11] locally, *i.e.,* within the computation of $F^\sharp$. Granger observes that the concrete operator F often involves lower closure operators, *i.e.,* operators ρ that are monotonic, idempotent ($\rho \circ \rho = \rho$) and reductive ($\rho(X) \sqsubseteq X$). Given any sound abstraction $\rho^\sharp$ of

ρ, the limit $Y^\sharp_\delta$ of the sequence $Y^\sharp_0 = X^\sharp$, $Y^\sharp_{i+1} = Y^\sharp_i \triangle \rho^\sharp(Y^\sharp_i)$ is an abstraction of $\rho(\gamma(X^\sharp))$. Whenever $\rho^\sharp$ is not an optimal abstraction of ρ, $Y^\sharp_\delta$ may be significantly more precise than $\rho^\sharp(X^\sharp)$. A relevant application is the analysis of complex test conjunctions $C_1 \wedge \cdots \wedge C_p$ where each atomic test C_i is modeled in the abstract as $\rho^\sharp_i$. Generally, $\rho^\sharp = \rho^\sharp_1 \circ \cdots \circ \rho^\sharp_p$ is not optimal, even when each $\rho^\sharp_i$ is. A complementary application is the analysis of a single test C_i using a sequence of relaxed, non-optimal test abstractions. For instance, non-linear expression parts may be replaced with intervals computed based on variable bounds [17]. As applying the relaxed test refines these bounds, the relaxation is not idempotent and benefits from local iterations. The link between local iterations and least fixpoint refinements lies in the observation that $\rho(X)$ computes a trivial fixpoint: the greatest fixpoint of ρ smaller than X: $\mathrm{gfp}_X\, \rho$. In both cases, a decreasing iteration starts from an abstraction of a fixpoint ($\mathrm{lfp}_\perp\, F$ in one case, $\mathrm{gfp}_X\, \rho$ in the other) and computes a smaller abstraction of that fixpoint.

On Narrowings. While a lot of work has been devoted to designing smart widenings, narrowings have gathered far less attention. Some major domains, such as polyhedra [9], do not feature any. This may be explained by three facts: firstly, narrowings (unlike widenings) are not necessary to achieve soundness; secondly, performing a bounded number of decreasing iterations without narrowing is sometimes sufficient to recover enough precision after widening [5]; thirdly, when this simple technique is not sufficient, narrowings do not actually help further in practice and solutions beyond decreasing iterations must be considered [12]. In the following, we argue that Constraint Programming can be seen as a form of decreasing iteration, but uses different techniques that are, in some respects, more advanced than the corresponding ones used in Abstract Interpretation.

2.2 Constraint Programming

We now present the basic definitions of Constraint Programming (see [21] for a more detailed presentation). In this section, we employ CP terminology, and take special care to point out terms with a different meaning in AI and CP.

Problems are modeled in a specific format, called Constraint Satisfaction Problem (CSP), and defined as follows:

Definition 1 (Constraint Satisfaction Problem). *A CSP is defined by a set of variables* $(v_1, \ldots, v_n)$ *taking their value in domains* $(\hat{D}_1, \ldots, \hat{D}_n)$ *and a set of constraints* $(C_1, \ldots, C_p)$ *that are relations on the variables.*

A *domain* D_i in CP denotes the set of possible values for a variable v_i and $D = D_1 \times \cdots \times D_n$ is called the *search space*. As the search space evolves during the solving process, we distinguish the initial search space of the CSP and note it $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ as in Def. 1. Problems may be discrete ($\hat{D} \subseteq \mathbb{Z}^n$) or continuous ($\hat{D} \subseteq \mathbb{R}^n$). Domains are, however, always bounded.

Given a constraint C on variables $v_1, \ldots, v_n$ in domains $D_1, \ldots, D_n$, and given values $x_i \in D_i$, we denote by $C(x_1, \ldots, x_n)$ the fact that the constraint is satified when each variable v_i takes the value x_i. The set of solutions is $S = \{(s_1, \ldots, s_n) \in \hat{D} \,|\, \forall i \in [\![1, p]\!], C_i(s_1, \ldots, s_n)\}$, with p the number of constraints and where $[\![a, b]\!] = \{x \in \mathbb{Z} \,|\, a \leq x \leq b\}$ denotes the interval of integers between a and b.

For discrete problems, two domain representations are traditionally used: subsets and intervals.

Definition 2 (Integer Cartesian Product). *Let $v_1, \ldots, v_n$ be variables over finite discrete domains $\hat{D}_1, \ldots, \hat{D}_n$. We call integer Cartesian product any Cartesian product of integer sets in $\hat{D}$. Integer Cartesian products form a finite lattice:*

$$\mathcal{S}^\sharp = \{ \prod_i X_i \,|\, \forall i,\, X_i \subseteq \hat{D}_i \}$$

Definition 3 (Integer Box). *Let $v_1, \ldots, v_n$ be variables over finite discrete domains $\hat{D}_1, \ldots, \hat{D}_n$. We call integer box a Cartesian product of integer intervals in $\hat{D}$. Integer boxes form a finite lattice:*

$$\mathcal{I}^\sharp = \{ \prod_i [\![a_i, b_i]\!] \,|\, \forall i,\, [\![a_i, b_i]\!] \subseteq \hat{D}_i,\, a_i \leq b_i \} \cup \{\emptyset\}$$

For continuous problems, domains are represented as intervals with floating-point bounds. Let $\mathbb{F}$ be the set of floating-point machine numbers. Given $a, b \in \mathbb{F}$, we note $[a, b] = \{x \in \mathbb{R} \,|\, a \leq x \leq b\}$ the interval of reals bounded by a and b, and $\mathbb{I} = \{[a, b] \,|\, a, b \in \mathbb{F}\}$ the set of such intervals.

Definition 4 (Box). *Let $v_1, \ldots, v_n$ be variables over bounded continuous domains $\hat{D}_1, \ldots, \hat{D}_n \in \mathbb{I}$. A box is a Cartesian product of intervals in $\hat{D}$. Boxes form a finite lattice:*

$$\mathcal{B}^\sharp = \{\prod_i I_i \,|\, \forall i,\, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i\} \cup \{\emptyset\}$$

Solving a CSP means computing exactly or approximating its solution set S.

Definition 5 (Approximation). *A complete (resp. sound) approximation of the solution S is a collection $\mathcal{A}$ of domain sequences such that $\forall (D_1, \ldots, D_n) \in \mathcal{A}$, $\forall i$, $D_i \subseteq \hat{D}_i$ and $S \subseteq \bigcup_{(D_1,\ldots,D_n)\in\mathcal{A}} D_1 \times \cdots \times D_n$ (resp. $\bigcup_{(D_1,\ldots,D_n)\in\mathcal{A}} D_1 \times \cdots \times D_n \subseteq S$).*

Soundness guarantees that we find only solutions, while completeness guarantees that no solution is lost. On discrete domains, constraint solvers are expected to be sound and complete, *i.e.,* compute the exact set of solutions. This is generally impossible on continuous domains, and we usually withdraw either soundness

(most of the time) or completeness. Note that the terms *sound* and *complete* have opposing definitions in AI and CP so, to avoid confusion, we will use the term *over-approximations* (resp. *under-approximations*) to denote CP-complete AI-sound (resp. CP-sound AI-complete) approximations.

In this article, we consider solving methods that over-approximate the solutions of continuous problems and compute the exact solutions of discrete ones. These methods alternate two steps: propagation and search.

Propagation. The goal of a propagation algorithm is to use the constraints to reduce the domains. Intuitively, we remove inconsistent values from domains, *i.e.,* values that cannot appear in any solution. Several definitions of consistency have been proposed in the literature. We present the most common ones.

Definition 6 (Generalized Arc-Consistency). *Given variables* $v_1, \ldots, v_n$ *over finite discrete domains* $D_1, \ldots, D_n$, $D_i \subseteq \hat{D}_i$, *the domains are said generalized arc-consistent (GAC) for a constraint* C *iff* $\forall i \in [\![1, n]\!], \forall x_i \in D_i, \forall j \neq i, \exists x_j \in D_j$ *such that* $C(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$ *holds.*

Definition 7 (Bound-Consistency). *Given variables* $v_1, \ldots, v_n$ *over finite discrete domains* $D_1, \ldots, D_n$, $D_i \subseteq \hat{D}_i$, *the domains are said bound-consistent (BC) for a constraint* C *iff* $\forall i \in [\![1, n]\!]$, D_i *is an integer interval* $[\![a_i, b_i]\!]$, *and the condition of Def. 6 holds for* $x_i = a_i$ *and* $x_i = b_i$ *(but not necessarily other values of* x_i *in* $[\![a_i, b_i]\!]$*).*

Definition 8 (Hull-Consistency). *Given variables* $v_1, \ldots, v_n$ *over continuous interval domains* $D_1, \ldots, D_n \in \mathbb{I}$, $D_i \subseteq \hat{D}_i$, *the domains are said Hull-consistent for a constraint* C *iff* $D_1 \times \cdots \times D_n$ *is the smallest floating-point box containing all the solutions for* C *in* $D_1 \times \cdots \times D_n$.

Each constraint kind and consistency comes with an algorithm, called *propagator*, that tries to achieve consistency. When considering several constraints, a *propagation loop* iterates the constraint propagators until a fixpoint is reached. As shown in [2], the order of propagator applications does not matter, as the set of domains lives in a finite lattice ($\mathcal{B}^\sharp$, $\mathcal{I}^\sharp$ or $\mathcal{S}^\sharp$) and the consistent fixpoint is its unique least element. When consistency is too costly to achieve, the propagators and propagator loops settle instead for an over-approximation (*e.g.,* removing only some inconsistent values). In addition to providing a tighter search space, the propagation is sometimes able to discover that it contains no solution at all, or that all its points are solutions.

Search. Generally, propagation alone cannot compute the exact solution (in the discrete case) or a precise enough over-approximation (in the continuous case). Thus, in a second step, a *search engine* is employed to try various assumptions on variable values. In the discrete case, a chosen variable is instantiated to each value in its domain. In the continuous case, its domain is split into two smaller subdomains. The solving algorithm continues by selecting a new search space and

```
list of boxes sols ← ∅                                  /*stores the solutions*/
queue of boxes toExplore ← ∅                            /*stores the boxes to explore*/
push D̂ in toExplore                                     /*initialization with CSP search space*/

while toExplore ≠ ∅ do
    b ← pop(toExplore)
    b ← Hull-consistency(b)
    if b ≠ ∅ then
        if b contains only solutions or b is small enough then
            sols ← sols ∪ b
        else
            split b into b1 and b2 by cutting in half along the largest box dimension
            push b1 and b2 in toExplore
```

Fig. 1. A classic continuous solver

applying a propagation step (as it may no longer be consistent), and possibly making further choices. This interleaving of propagations and choices terminates when the search space can be proved to contain no solution, only solutions or, in the continuous case, when its size is below a user-specified threshold. In the discrete case, at worst, all the variables are instantiated. After exploring a branch, in case of failure or if all the solutions should be computed, the algorithm returns to a choice point (instantiation or split) by *backtracking* and tries another assumption.

We illustrate the search algorithm by an example solver in Fig. 1 corresponding to a continuous solver based on Hull-Consistency (Def. 8) computing an over-approximation of all the solutions. As explained above, a discrete solver would differ significantly. Existing solutions to embed discrete variables in continuous solvers consist in adding constraints expressing integerness and their propagators [6,4], while keeping a search engine based on continuous domains.

2.3 Comparing Abstract Interpretation and Constraint Programming

We now present informally some connections between Abstract Interpretation and Constraint Programming. The next section will make these connections formal by expression CP in the AI framework.

Both techniques are grounded in the theory of fixpoints in lattices. They pursue similar goals and means: computing or over-approximating solutions to complex equations by manipulating abstracted views of potential solution sets, such as boxes (called domains in CP, and abstract domain elements in AI). Their goals, however, do not coincide. Solvers aim at completeness and thus always allow refinement up to an arbitrary precision. On the contrary, the precision of abstract interpreters is fixed by their choice of abstract domains; they can seldom

represent arbitrary precise over-approximations. AI embraces incompleteness. The choice of abstract domains sets the cost and precision of an interpreter, while the choice of domains sets the cost of a solver to reach a given precision.

Although they aim at completeness, solvers nevertheless employ simple, non-relational domains. They rely on collections of simple domains (similar to disjunctive completions) to reach the desired precision. The domains are homogeneous and cannot mix variables of different type. On the contrary, AI enjoys a rich collection of abstract domains, including relational and heterogeneous ones.

On the algorithmic side, AI and CP share common ideas. Iterated propagations in CP are similar to local iterations in AI. In fact, approximating consistency in CP is similar to approximating the effect of a complex test in AI. However, search engines in CP use features, such as choice points and backtracking, that have no equivalent in AI. Dually, the widening from AI has no equivalent in CP, as CP does not employ increasing iterations but only decreasing ones.

Finally, while abstract interpreters are usually defined in a very generic way and parametrized by arbitrary abstract domains, solvers are far less flexible and embed choices of abstractions (such as domains and consistencies) as well as concrete semantics (the type of variables) in their design. In the following, we will design an abstract solver that avoids these pitfalls and can benefit from the large library of abstract domains designed for AI.

3 An Abstract Constraint Solver

We now present our main contribution: expressing constraint solving as an abstract interpreter, which involves defining concrete and abstract domains, abstract operators for split and consistency, and an iteration scheme.

3.1 Concrete Solving

A CSP is similar to the analysis of a conjunction of tests and can be formalized in terms of local iterations. We consider as concrete domain $\mathcal{D}$ the subsets of the CSP search space $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ (Def. 1), *i.e.*, $(\mathcal{P}(\hat{D}), \subseteq, \emptyset, \cup)$. Each constraint C_i corresponds to a concrete lower closure operator $\rho_i : \mathcal{P}(\hat{D}) \to \mathcal{P}(\hat{D})$, such that $\rho_i(X)$ keeps only the points in X satisfying C_i. The concrete solution of the problem is simply $S = \rho(\hat{D})$, where $\rho = \rho_1 \circ \cdots \circ \rho_p$. It is expressed in fixpoint form as $\mathrm{gfp}_{\hat{D}}\, \rho$.

3.2 Abstract Domains

Solvers do not manipulate individual points in $\hat{D}$, but rather collections of points of certain forms, such as boxes, called domains in CP. We now show that CP-domains are elements of an abstract domain $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$ in AI, which depends on the chosen consistency. In addition to standard AI operators, we require a monotonic *size function* $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$ that we will use later as a termination criterion (Def. 10).

Example 1. Generalized arc-consistency (Def. 6) corresponds to the abstract domain of integer Cartesian products $\mathcal{S}^\sharp$ (Def. 2), ordered by element-wise set inclusion. It is linked with the concrete domain $\mathcal{D}$ by the standard Cartesian Galois connection:

$$\begin{aligned}
&\mathcal{D} \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} \mathcal{S}^\sharp \\
&\gamma_a(S_1, \ldots, S_n) = S_1 \times \cdots \times S_n \\
&\alpha_a(X) = \lambda i.\{x \,|\, \exists (x_1, \ldots, x_n) \in X,\, x_i = x\}
\end{aligned}$$

The size function τ_a uses the size of the largest component, minus one, so that singletons have size 0:

$$\tau_a(S_1, \ldots, S_n) = \max_i(|S_i| - 1)$$

Example 2. Bound consistency (Def. 7) corresponds to the domain of integer boxes $\mathcal{I}^\sharp$ (Def. 3), ordered by element-wise interval inclusion. We have a Galois connection, and use as size function the length of the largest dimension:

$$\begin{aligned}
&\mathcal{D} \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} \mathcal{I}^\sharp \\
&\gamma_b([\![a_1, b_1]\!], \ldots, [\![a_n, b_n]\!]) = [\![a_1, b_1]\!] \times \cdots \times [\![a_n, b_n]\!] \\
&\alpha_b(X) = \lambda i.[\![\min\{x \in \mathbb{Z} \,|\, \exists (x_1, \ldots, x_n) \in X,\, x_i = x\}, \\
&\qquad\qquad \max\{x \in \mathbb{Z} \,|\, \exists (x_1, \ldots, x_n) \in X,\, x_i = x\}]\!] \\
&\tau_b([\![a_1, b_1]\!], \ldots, [\![a_n, b_n]\!]) = \max_i(b_i - a_i)
\end{aligned}$$

Example 3. Hull consistency (Def. 8) corresponds to the domain of boxes with floating-point bounds $\mathcal{B}^\sharp$ (Def. 4). We use the following Galois connection and size function:

$$\begin{aligned}
&\mathcal{D} \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \mathcal{B}^\sharp \\
&\gamma_h([a_1, b_1], \ldots, [a_n, b_n]) = [a_1, b_1] \times \cdots \times [a_n, b_n] \\
&\alpha_h(X) = \lambda i.[\max\{x \in \mathbb{F} \,|\, \forall (x_1, \ldots, x_n) \in X,\, x_i \geq x\}, \\
&\qquad\qquad \min\{x \in \mathbb{F} \,|\, \forall (x_1, \ldots, x_n) \in X,\, x_i \leq x\}] \\
&\tau_h([a_1, b_1], \ldots, [a_n, b_n]) = \max_i(b_i - a_i)
\end{aligned}$$

We observe that to each choice corresponds a classic non-relational abstract domain, which is an homogeneous Cartesian product of identical single-variable domains. However, this needs not be the case: new solvers can be designed beyond the ones considered in traditional CP by varying the abstract domains further. A first idea is to apply different consistencies to different variables which permits, in particular, mixing variables with discrete domains and variables with continuous domains. A second idea is to parametrize the solver with other abstract domains from the AI literature, in particular relational domains, which we illustrate below.

Example 4. The octagon domain $\mathcal{O}^\sharp$ [18] assigns a (floating-point) upper bound to each binary unit expression $\pm v_i \pm v_j$ on the variables $v_1, \ldots, v_n$. It enjoys a Galois connection, and we use the size function from [20]:

$$\mathcal{D} \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathcal{O}^\sharp$$
$$\mathcal{O}^\sharp = \{\alpha v_i + \beta v_j \,|\, i, j \in [\![1, n]\!],\, \alpha, \beta \in \{-1, 1\}\,\} \to \mathbb{F}$$
$$\gamma_o(X^\sharp) = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \,|\, \forall i, j, \alpha, \beta,\, \alpha x_i + \beta x_j \leq X^\sharp(\alpha v_i + \beta v_j)\}$$
$$\alpha_o(X) = \lambda(\alpha v_i + \beta v_j).\, \min\{x \in \mathbb{F} \,|\, \forall (x_1, \ldots, x_n) \in X,\, \alpha x_i + \beta x_j \leq x\}$$
$$\tau_o(X^\sharp) = \min(\, \max_{i,j,\beta} (X^\sharp(v_i + \beta v_j) + X^\sharp(-v_i - \beta v_j)),\ \max_i (X^\sharp(v_i + v_i) + X^\sharp(-v_i - v_i))/2)$$

Example 5. The polyhedron domain $\mathcal{P}^\sharp$ [9] abstract sets as convex, closed polyhedra. Modern implementations [15] generally follow the "double description approach" and maintain two dual representations for each polyhedron: a set of linear constraints and a set of generators (vertices and rays, although our polyhedra never feature rays as they are bounded). There is no abstraction function α for polyhedra, and so, no Galois connection. Operators are generally easier on one representation. In particular, we define the size function on generators as the maximal Euclidian distance between pairs of vertices:

$$\tau_p(X^\sharp) = \max_{g_i, g_j \in X^\sharp} ||g_i - g_j||$$

3.3 Constraints and Consistency

We now assume that an abstract domain $\mathcal{D}^\sharp$ underlying the solver is fixed. Given the concrete semantics of the constraints $\rho = \rho_1 \circ \cdots \circ \rho_p$, and if $\mathcal{D}^\sharp$ enjoys a Galois connection $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, then the semantics of the perfect propagator achieving the consistency for all the constraints is simply: $\alpha \circ \rho \circ \gamma$. Solvers achieve this algorithmically by applying the propagator for each constraint in turn until a fixpoint is reached or, when this process is deemed too costly, return before a fixpoint is reached. By observing that each propagator corresponds to an abstract test transfer function $\rho_i^\sharp$ in $\mathcal{D}^\sharp$, we retrieve the local iterations proposed by Granger to analyze conjunctions of tests [11]. A trivial narrowing is used here: stop refining after an iteration limit is reached.

Additionally, each $\rho_i^\sharp$ can be internally implemented by local iterations [11], a technique which is used in both the AI and CP communities. A striking connection is the analysis in non-relation domains using forward-backward iterations on expression trees [17, §2.4.4], which is extremely similar to the HC4-revise algorithm [3] developed independently for CP.

When there is no Galois connections (as for polyhedra), there is no equivalent to consistency. Nevertheless, we can still use local iterations on approximate test transfer functions $\rho_i^\sharp$, which serve the same purpose: to remove some points that do not satisfy the constraints.

3.4 Disjunctive Completion and Split

In order to approximate the solution to an arbitrary precision, solvers use a coverage of finitely many abstract elements from $\mathcal{D}^\sharp$. This corresponds in AI to the notion of disjunctive completion. We now consider the abstract domain $\mathcal{E}^\sharp = \mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$, and equip it with the Smyth order $\sqsubseteq^\sharp_\mathcal{E}$, a classic order for disjunctive completions defined as:

$$X^\sharp \sqsubseteq^\sharp_\mathcal{E} Y^\sharp \iff \forall B^\sharp \in X^\sharp, \exists C^\sharp \in Y^\sharp, B^\sharp \sqsubseteq^\sharp C^\sharp$$

The creation of new disjunctions is achieved by a split operation $\oplus$, that splits an abstract element into two or more elements:

Definition 9 (Split Operator). *A* split operator $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ *satisfies:*

1. $\forall e \in \mathcal{D}^\sharp, |\oplus(e)|$ *is finite,*
2. $\forall e \in \mathcal{D}^\sharp, \forall e_i \in \oplus(e), e_i \sqsubseteq^\sharp e$, *and*
3. $\forall e \in \mathcal{D}^\sharp, \gamma(e) = \bigcup \{\gamma(e_i) \,|\, e_i \in \oplus(e)\}$.

Condition 2 implies $\oplus(e) \sqsubseteq^\sharp_\mathcal{E} \{e\}$. Condition 3 implies that $\oplus$ is an abstraction of the identity; thus, $\oplus$ can be freely applied at any place during the solving process without altering the AI-soundness (over-approximation). We now present a few example splits.

Example 6 (Split in $\mathcal{S}^\sharp$). The instantiation of a variable v_i in a discrete domain $X^\sharp = (S_1, \ldots, S_n) \in \mathcal{S}^\sharp$ is a split operator:

$$\oplus_a(X^\sharp) = \{(S_1, \ldots, S_{i-1}, \{x\}, S_{i+1}, \ldots, S_n) \,|\, x \in S_i\}$$

Example 7 (Split in $\mathcal{B}^\sharp$). Cutting a box in two along a variable v_i in a continuous domain $X^\sharp = (I_1, \ldots, I_n) \in \mathcal{B}^\sharp$ is a split operator:

$$\oplus_h(X^\sharp) = \{(I_1, \ldots, I_{i-1}, [a, h], I_{i+1}, \ldots, I_n),\ (I_1, \ldots, I_{i-1}, [h, b], I_{i+1}, \ldots, I_n)\}$$

where $I_i = [a, b]$ and $h = (a + b)/2$ rounded in $\mathbb{F}$ in any direction.

Example 8 (Split in $\mathcal{O}^\sharp$). Given a binary unit expression $\alpha v_i + \beta v_j$, we define the split on an octagon $X^\sharp \in \mathcal{O}^\sharp$ along this expression as:

$$\oplus_o(X^\sharp) = \{X^\sharp[(\alpha v_i + \beta v_j) \mapsto h],\ X^\sharp[(-\alpha v_i - \beta v_j) \mapsto -h]\}$$

where $h = (X^\sharp(\alpha v_i + \beta v_j) - X^\sharp(-\alpha v_i - \beta v_j))/2$, rounded in $\mathbb{F}$ in any direction.

Example 9 (Split in $\mathcal{P}^\sharp$). Given a polyhedron $X^\sharp \in \mathcal{P}^\sharp$ represented as a set of linear constraints, and a linear expression $\sum_i \beta_i v_i$, we define the split:

$$\oplus_p(X^\sharp) = \{X^\sharp \cup \{\textstyle\sum_i \beta_i v_i \leq h\}, X^\sharp \cup \{\sum_i \beta_i v_i \geq h\}\}$$

where $h = (\min_{\gamma(X^\sharp)} \sum_i \beta_i v_i + \max_{\gamma(X^\sharp)} \sum_i \beta_i v_i)/2$ can be computed by the Simplex algorithm.

These splits are parametrized by the choice of a direction of cut (some variable or expression). For non-relational domains we can use two classic strategies from CP: split each variable in turn, or split along a variable with maximal size (*i.e.,* $|S_i|$ or $b_i - a_i$). These strategies lift naturally to octagons by replacing the set of variables with the (finite) set of unit binary expressions (see also [20]). For polyhedra, one can bisect the segment between two vertices that are the farthest apart, in order to minimize τ_p. However, even for relational domains, we can use a faster and simpler non-relational split, *e.g.,* cut along the variable with the largest range.

To ensure the termination of the solver, we impose that any series of reductions, splits, and choices eventually outputs a small enough element for τ:

Definition 10. *The operators* $\tau : \mathcal{D}^\sharp \to \mathcal{R}^+$ *and* $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ *are compatible if, for any reductive operator* $\rho^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *(i.e.,* $\forall X^\sharp \in \mathcal{D}^\sharp, \rho^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$*) and any family of choice operators* $\pi_i : \mathcal{E}^\sharp \to \mathcal{D}^\sharp$ *(i.e.,* $\forall Y^\sharp \in \mathcal{E}^\sharp, \pi_i(Y^\sharp) \in Y^\sharp$*), we have:*

$$\forall e \in \mathcal{D}^\sharp, \forall r \in \mathbb{R}^{>0}, \exists K \text{ s.t. } \forall j \geq K, (\tau \circ \pi_j \circ \oplus \circ \rho \circ \cdots \circ \pi_1 \circ \oplus \circ \rho)(e) \leq r$$

Each of the split function we presented above, $\oplus_a$, $\oplus_h$, $\oplus_o$, and $\oplus_p$, is compatible with the size function τ_a, τ_h, τ_o, and τ_p we proposed on the corresponding domain.

The search procedure can be represented as a search tree where each node corresponds to a search space and the children of a node are constructed by applying the split operator on the parent and then applying a reduction. With this representation, the set of nodes at a given depth corresponds to a disjunction over-approximating the solution. Moreover, a series of reduction (ρ), selection (π), and split ($\oplus$) operators corresponds to a tree branch. Definition 10 states that each branch of the search tree is finite.

3.5 Abstract Solving

We are now ready to present our solving algorithm, in Fig. 2. It maintains in toExplore and sols two disjunctions in $\mathcal{E}^\sharp$, and iterates the following steps: choose an abstract element e from toExplore (**pop**), apply the consistency ($\rho^\sharp$), and either discard the result, add it to the set of solutions sols, or split it ($\oplus$). The solver starts with the maximal element $\top^\sharp$ of $\mathcal{D}^\sharp$, which represents $\gamma(\top^\sharp) = \hat{D}$.

Correctness. At each step, $\bigcup\{\gamma(x) \mid x \in \text{toExplore} \cup \text{sols}\}$ is an over-approximation of the set of solutions, because the consistency $\rho^\sharp$ is an abstraction of the concrete semantics ρ of the constraints and the split $\oplus$ is an abstraction of the identity. We note that abstract elements in sols are consistent and either contain only solutions or are smaller than r. The algorithm terminates when toExplore is empty, at which point sols over-approximates the set of solutions with consistent elements that contain only solutions or are smaller than r. To compute the exact set of solutions in the discrete case, it is sufficient to choose $r < 1$.

```
list of abstract domains sols ← ∅                    /*stores the abstract solutions*/
queue of abstract domains toExplore ← ∅ /*stores the abstract elements to explore*/
push ⊤♯ in toExplore /*initialization with the abstract search space: γ(⊤♯) = D̂ */

while toExplore ≠ ∅ do
    e ← pop(toExplore)
    e ← ρ♯(e)
    if e ≠ ∅ then
        if τ(e) ≤ r or isSol(e) /*isSol(e) returns true if e contains only solutions*/
        then
            sols ← sols ∪ e
        else
            push ⊕(e) in toExplore
```

Fig. 2. Our generic abstract solver

The termination is ensured by the following proposition:

Proposition 1. *If τ and $\oplus$ are compatible, the algorithm in Fig. 2 terminates.*

Proof. The search tree is finite. Otherwise, as its width is finite by Def. 9, there would exist an infinite branch (König's lemma), which would contradict Def. 10.

The solver in Fig. 2 uses a queue data-structure, and splits the oldest abstract element first. More clever choosing strategies are possible (*e.g.,* split the largest element for τ). The algorithm remains correct and terminates for any strategy.

Comparison with Abstract Interpretation. Similarly to local iterations in AI, our solver performs decreasing abstract iterations. Indeed, toExplore $\cup$ sols is decreasing for $\sqsubseteq^\sharp_{\mathcal{E}}$ in the disjunctive completion domain $\mathcal{E}^\sharp$ at each iteration of the loop (indeed, $\oplus(e) \sqsubseteq^\sharp_{\mathcal{E}} \{e\}$ and we can assume that $\rho^\sharp$ is reductive in $\mathcal{D}^\sharp$ without loss of generality). However, it differs from classic AI in two ways. Firstly, there is no split operator in AI: new components in a disjunctive completion are generally added only at control-flow joins (by delaying the abstract join $\sqcup^\sharp$ in $\mathcal{D}^\sharp$). Secondly, the solving iteration strategy is far more elaborated than in AI. The use of a narrowing is replaced with a data-structure that maintains an ordered list of abstract elements and a splitting strategy that performs a refinement process and ensures its termination. Actually, more complex strategies than the simple one we presented here exist in the CP literature. One example is the AC-5 algorithm [26] where, each time the domain of a variable changes, the variable decides which constraints need to be propagated. The design of efficient propagation algorithms is an active research area [22].

4 Experiments

We have implemented a prototype solver to demonstrate the feasibility of our approach. We describe its main features and present experimental results.

4.1 Implementation

Our prototype solver, called Absolute, is implemented in OCaml. It uses Apron, a library of numeric abstract domains intended primarily for static analysis [15]. We benefit from Apron domains (intervals, octagons, and polyhedra), its ability to hide their internal algorithms under a uniform API, and its handling of integer and real variables and of non-linear constraints.

Consistency. Apron provides a language of constraints sufficient to express many CSPs: equalities and inequalities over numeric expressions (including operators such as $+$, $-$, $\times$, $/$, $\sqrt{}$, power, modulo, and rounding to integers). The test transfer function naturally provides propagators for these constraints. Internally, each domain implements its own algorithm to handle tests, including sophisticated methods to handle non-linear constraints (such as HC4 and linearization [17]). Our solver then performs local iterations until either a fixpoint or a maximum number of iterations is reached (which is set to 3 to ensure a fast solving). In CP solvers, only the constraints containing at least one variable that has been modified during the previous step are propagated. However, for simplicity, our solver propagates all the constraints at each step of the local iteration.

Split. Currently, our solver only splits along a single variable at a time, cutting its range in two, even for relational domains and integer variables. It chooses the variable with the largest range. It uses a queue to maintain the set of abstract elements to explore (as in Fig. 2). Compared to most CP solvers, this splitting strategy is very basic. It will be improved in the future by integrating more clever strategies from the CP literature.

4.2 Exemple of AI-Solving with Absolute

In order to make the abstract solving process clear, we detail here an example with a very simple problem. Consider a CSP on two continuous variables v_1 and v_2 taking their values in $D_1 = D_2 = [-5, 5]$, and the constraints $C_1 : x^2 + y^2 \leq 4$ and $C_2 : (x-2)^2 + (y+1)^2 \leq 1$.

Figure 3 shows the first iterations of the AI-solving method for this CSP. The root corresponds to the initial search space after applying the reduction based on HC4 as explained above ($D_1 = [1, 2], D_2 = [-1.73, 0]$). Its successor nodes correspond to the search spaces obtained after splitting the domain D_2 in half and applying the reduction to the new states. These two steps (split and reduction) are repeatedly applied until all the solutions have been found, but Figure 3 only shows the first three steps.

At a given depth in the search tree, the current approximation of the solution space is made of the disjunction of the abstract elements currently investigated. Figure 4 shows these disjunctions for the search tree depicted in Fig. 3.

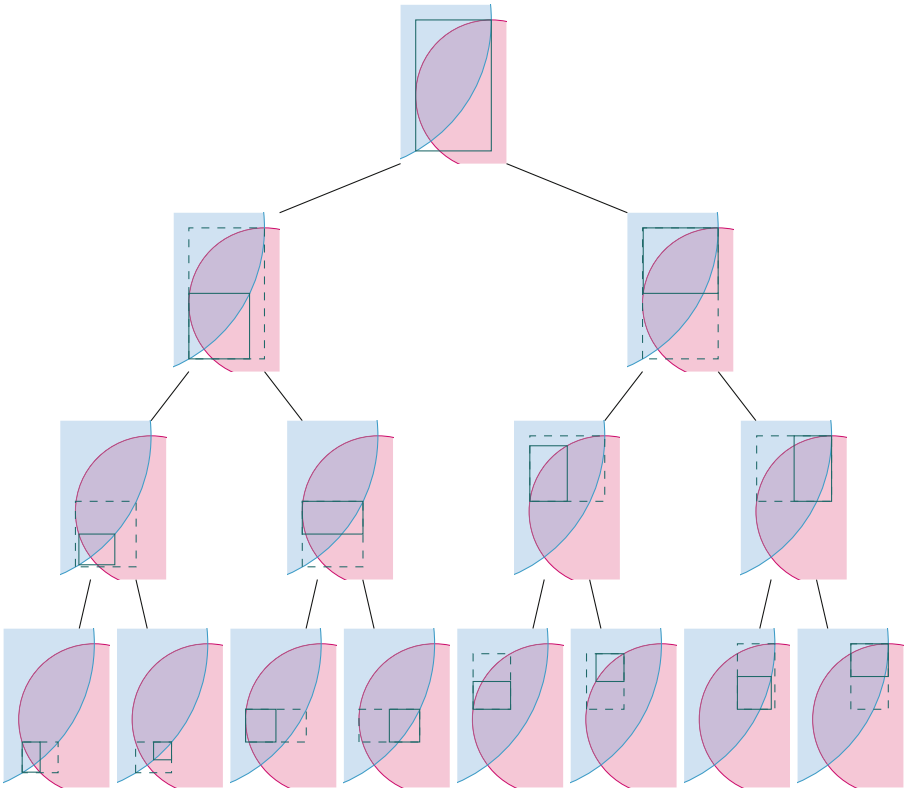


Fig. 3. First iterations of the AI-solving method

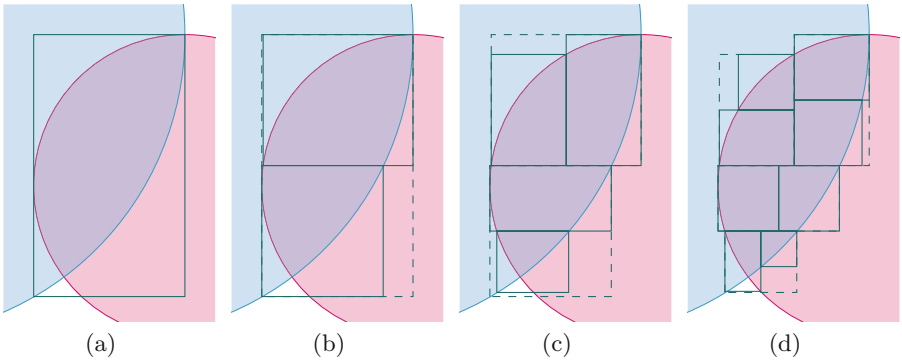


Fig. 4. Disjunctions of the first iterations of the AI-solving method search tree given in Fig. 3

Table 1. CPU time in seconds to find the first solution with Ibex and Absolute

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	=	0.009	0.018	0.053	0.048
nbody5.1	6	=	32.85	708.47	0.027	$\geq$ 1h
ipp	8	=	0.66	9.64	19.28	1.46
brent-10	10	=	7.96	4.57	0.617	$\geq$ 1h
KinematicPair	2	$\leq$	0.013	0.018	0.016	0.011
biggsc4	4	$\leq$	0.011	0.022	0.096	0.029
o32	5	$\leq$	0.045	0.156	0.021	0.263

Table 2. CPU time in seconds to find all solutions with Ibex and Absolute

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	=	0.02	0.10	0.26	0.14
nbody5.1	6	=	95.99	1538.25	27.08	$\geq$ 1h
ipp	8	=	38.83	39.24	279.36	817.86
brent-10	10	=	21.58	263.86	330.73	$\geq$ 1h
KinematicPair	2	$\leq$	59.04	23.14	60.78	31.11
biggsc4	4	$\leq$	800.91	414.94	1772.52	688.56
o32	5	$\leq$	27.36	22.66	40.74	33.17

4.3 Experimental Results

We have run Absolute on two classes of problems: firstly, on continuous problems to compare its efficiency with state-of-the-art CP solvers; secondly, on mixed problems, that these CP solvers cannot handle while our abstract solver can.

Continuous Solving. We use problems from the COCONUT benchmark[2], a standard CP benchmark with only real variables. We compare Absolute with the standard (interval-based) Ibex CP continuous solver[3]. Notice that the COCONUT problems have a relatively small number of variables, compared for instance to the number of variables that can be analyzed for a single program in AI. The difficulty of the benchmark is here due to both the expressions of the constraints (non linear with multiple variable occurences) and the high precision that is required.

Additionally, we compare Absolute to our extension of Ibex to octagons from previous work [20], which allows comparing the choice of domain (intervals versus octagons) independently from the choice of solver algorithm (classic CP solver versus our AI-based solver). Tables 1 and 2 show the run time in seconds to find

[2] Available at http://www.mat.univie.ac.at/~neum/glopt/coconut/
[3] Available at http://www.emn.fr/z-info/ibex/

Table 3. Number of nodes created to find the first solution with Ibex and Absolute

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	=	145	28	45	207
nbody5.1	6	=	262 659	2 765 630	105	-
ipp	8	=	4 039	25 389	899	3 421
brent-10	10	=	101 701	12 744	2 113	-
KinematicPair	2	$\leq$	43	55	39	55
biggsc4	4	$\leq$	98	96	94	84
o32	5	$\leq$	87	344	85	942

Table 4. Number of nodes created to find all solutions with Ibex and Absolute

			$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
name	# vars	ctr type	Ibex	Absolute	Ibex	Absolute
b	4	=	551	577	147	1057
nbody5.1	6	=	598 521	5 536 283	7 925	-
ipp	8	=	237 445	99 179	39 135	2 884 925
brent-10	10	=	211 885	926 587	5 527	-
KinematicPair	2	$\leq$	847 643	215 465	520 847	215 465
biggsc4	4	$\leq$	3 824 249	6 038 844	2 411 741	6 037 260
o32	5	$\leq$	161 549	120 842	84 549	111 194

all the solutions or only the first solution of each problem. Tables 3 and 4 show the number of nodes created to find all the solutions or only the first solution of each problem.

On average, Absolute is competitive with the traditional CP approach. More precisely, it is globally slower on problems with equalities, and faster on problems with inequalities. This difference of performance seems to be related to the following ratio: the number of constraints in which a variable appears over the total number of constraints. As said previously, at each iteration, all the constraints are propagated even those for which none of their variables have changed. This increases the computation time at each step and thus increases the overall time. For instance, in the problem `brent-10`, there are ten variables, ten constraints, and each variable appears in at most three constraints. If only one variable has been modified, we will nevertheless propagate all ten constraints, instead of three at most. This may explain the timeouts observed on problems `brent-10` and `nbody5.1` with Absolute.

Moreover, in our solver, the consistency loop is stopped after three iterations while, in the classic CP approach, the fixpoint is reached. The consistency in Absolute may be less precise than the one used in Ibex, which reduces the time spent during the propagation step but may increase the search phase. This probably explains why in tables 3 and 4 the number of nodes created during the

Table 5. CPU time, in seconds, to solve mixed problems with Absolute

name	# vars		ctr type	First Solution			All solutions		
	int	real		$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$	$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$
gear4	4	2	$=$	0.016	0.036	0.296	0.017	0.048	0.415
st_miqp5	2	5	$\leq$	0.672	1.152	$\geq$ 1h	2.636	3.636	$\geq$ 1h
ex1263	72	20	$= \leq$	8.747	$\geq$ 1h	$\geq$ 1h	473.933	$\geq$ 1h	$\geq$ 1h
antennes_4_3	6	2	$\leq$	3.297	22.545	$\geq$ 1h	520.766	1562.335	$\geq$ 1h

Table 6. Number of nodes created to solve mixed problems with Absolute

name	# vars		ctr type	First Solution			All solutions		
	int	real		$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$	$\mathcal{B}^\sharp$	$\mathcal{O}^\sharp$	$\mathcal{P}^\sharp$
gear4	4	2	$=$	43	226	226	67	501	501
st_miqp5	2	5	$\leq$	2 247	2 247	-	7 621	7 621	-
ex1263	72	20	$= \leq$	8544	-	-	493 417	-	-
antennes_4_3	6	2	$\leq$	17 625	40 861	-	2 959 255	6 657 237	-

solving process with Absolute is most of the times larger than the one with Ibex. Less reductions are performed thus more splitting operations are needed, hence more nodes are created during the solving process.

These experimentations show that our prototype, which only features quite naïve CP strategies, behaves reasonably well on a classic benchmark. Further studies will include a deeper analysis of the performances and improvements of Absolute on its identified weaknesses (splitting strategy, propagation loop).

Mixed Discrete-Continuous Solving. As CP solvers seldom handle mixed problems, no standard benchmark exists. We thus gathered problems from MINLPLib,[4] a library of mixed optimisation problems from the Operational Research community. These problems are not satisfaction CSPs, but optimization problems, with constraints to satisfy and a function to minimize. We thus needed to turn them info satisfaction CSPs. Following the approach in [4], we replaced each optimization criterion $\min f(x)$ with a constraint $|f(x) - \text{best\_known\_value}| \leq \epsilon$. We compared Absolute to the mixed solving scheme from [4], using the same ϵ and benchmarks, and found that they have similar run times (we do not provide a more detailed comparison as it would be meaningless due to the machine differences).

More interestingly, we observe that Absolute can solve mixed problems in reasonable time and behaves better with intervals than with relational domains. A possible reason is that current propagations and heuristics are not able to fully use relational information available in octagons or polyhedra. Previous

[4] Available at `http://www.gamsworld.org/minlp/minlplib.htm`

works [20] suggest that a carefully designed split is key to efficient octagons; future work will incorporate ideas from [20] into our solver and develop them further. Already, Absolute is able to naturally cope with mixed CP problems in a reasonable time, opening the way to new CP applications such as robotic localization [14] or geometric problems [1].

5 Conclusion

In this paper, we have exposed some links between AI and CP, and used them to design a CP solving scheme built entirely on abstract domains. The preliminary results obtained with our prototype are encouraging and open the way to the development of hybrid CP–AI solvers able to naturally handle mixed constraint problems. In future work, we wish to improve our solver by adapting and integrating advanced methods from the CP literature. The areas of improvement include: split operators for abstract domains, specialized propagators (such as octagonal consistency or global constraints), and improvements to the propagation loop. We built our solver on abstractions in a modular way, so that existing and new methods can be combined together, as is the case for reduced products in AI. Ultimately, each problem should be automatically solved in the abstract domains which best fit it, as it is the case in AI. A natural future work is thus the development of new abstract domains adapted to specific constraint kinds. Another exciting development would be to use some methods form CP in an AI-based static analyzer. Areas of interest include: decreasing iteration methods, which are more advanced in CP than in AI, the use of a split operator in disjunctive completion domains, and the ability of CP to refine an abstract element to achieve completeness. Finally, it also remains to understand how fixpoint extrapolation operators, such as widenings, which are very popular in AI, can be exploited in CP solvers.

References

1. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
2. Benhamou, F.: Heterogeneous Constraint Solvings. In: Hanus, M., Rodríguez-Artalejo, M. (eds.) ALP 1996. LNCS, vol. 1139, pp. 62–76. Springer, Heidelberg (1996)
3. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.-F.: Revisiting hull and box consistency. In: Proc. of the 16th Int. Conf. on Logic Programming, pp. 230–244 (1999)
4. Berger, N., Granvilliers, L.: Some interval approximation techniques for MINLP. In: SARA (2009)
5. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace 2010. AIAA (2010)

6. Chabert, G., Jaulin, L., Lorca, X.: A Constraint on the Number of Distinct Vectors with Application to Localization. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 196–210. Springer, Heidelberg (2009)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Rec. of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252. ACM Press (1977)
8. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 84–96 (1978)
10. D'Silva, V., Haller, L., Kroening, D.: Satisfiability Solvers Are Static Analysers. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 317–333. Springer, Heidelberg (2012)
11. Granger, P.: Improving the Results of Static Analyses of Programs by Local Decreasing Iterations. In: Shyamasundar, R.K. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 68–79. Springer, Heidelberg (1992)
12. Halbwachs, N., Henry, J.: When the Decreasing Sequence Fails. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 198–213. Springer, Heidelberg (2012)
13. Hervieu, A., Baudry, B., Gotlieb, A.: Pacogen: Automatic generation of pairwise test configurations from feature models. In: Proc. of the 22nd Int. Symposium on Software Reliability Engineering, pp. 120–129 (2011)
14. Jaulin, L., Bazeille, S.: Image shape extraction using interval methods. In: Proc. of the 15th IFAC Symposium on System Identification (2009)
15. 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)
16. Lazaar, N., Gotlieb, A., Lebbah, Y.: A CP framework for testing CP. Constraints 17(2), 123–147 (2012)
17. Miné, A.: Weakly Relational Numerical Abstract Domains. PhD thesis, École Polytechnique, Palaiseau, France (December 2004)
18. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
19. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Science 7(2), 95–132 (1974)
20. Pelleau, M., Truchet, C., Benhamou, F.: Octagonal Domains for Continuous Constraints. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 706–720. Springer, Heidelberg (2011)
21. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier (2006)
22. Schulte, C., Tack, G.: Implementing efficient propagation control. In: Proc. of the 3rd Workshop on Techniques for Implementing Constraint Programming Systems (2001)
23. Choco Team. Choco: an open source Java constraint programming library. Research report 10-02-INFO, École des Mines de Nantes (2010)

24. Thakur, A., Reps, T.: A Generalization of Stålmarck's Method. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 334–351. Springer, Heidelberg (2012)
25. Truchet, C., Pelleau, M., Benhamou, F.: Abstract domains for constraint programming, with the example of octagons. In: Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 72–79 (2010)
26. van Hentenryck, P., Deville, Y., Teng, C.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57 (1992)

An Abstract Interpretation of DPLL(T)*

Martin Brain[1], Vijay D'Silva[1], Leopold Haller[1],
Alberto Griggio[2,**], and Daniel Kroening[1]

[1] Computer Science Department, University of Oxford, Oxford, UK
first.last@cs.ox.ac.uk
[2] Fondazione Bruno Kessler, Trento, Italy
griggio@fbk.eu

Abstract. DPLL(T) is a central algorithm for Satisfiability Modulo Theories (SMT) solvers. The algorithm combines results of reasoning about the Boolean structure of a formula with reasoning about conjunctions of theory facts to decide satisfiability. This architecture enables modern solvers to combine the performance benefits of propositional satisfiability solvers and conjunctive theory solvers. We characterise DPLL(T) as an abstract interpretation algorithm that computes a product of two abstractions. Our characterisation allows a new understanding of DPLL(T) as an instance of an abstract procedure to combine reasoning engines beyond propositional solvers and conjunctive theory solvers. In addition, we show theoretically that the split into Boolean and theory reasoning is sometimes unnecessary and demonstrate empirically that it can be detrimental to performance.

1 Introduction

The previous decade has witnessed the development of efficient solvers for deciding satisfiability of formulae in a wide range of logical theories. The development of these *Satisfiability Modulo Theory* (SMT) solvers can be understood as a consequence of three advances. Two advances are improvements in the performance of solvers for Boolean satisfiability, and for the conjunctive fragments of first-order theories such as equality with uninterpreted functions [12], difference logic [20], or linear rational arithmetic [10]. The third advance is DPLL(T), an algorithm that efficiently combines the strengths of propositional SAT solvers and conjunctive theory solvers to decide satisfiability of a theory formula [12].

We explain the principles of DPLL(T) with an example. A satisfiability checker for the formula φ below has to reason about Boolean combinations of equality constraints.

$$\varphi \mathrel{\hat{=}} (x = y \vee y \neq z) \wedge x = z \wedge y = z \qquad \mathsf{BoolSkel}(\varphi) \mathrel{\hat{=}} (p \vee \neg q) \wedge r \wedge q$$

* Supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/J012564/1, and the FP7 STREP PINCETTE.
** Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 455–475, 2013.

A DPLL(T) solver first constructs a *Boolean skeleton* of φ, given as $\mathsf{BoolSkel}(\varphi)$ above. The Boolean skeleton has the same structure as φ, but does not include information about the theory. If $\mathsf{BoolSkel}(\varphi)$ is unsatisfiable, so is φ. If $\mathsf{BoolSkel}(\varphi)$ is satisfiable, each satisfying assignment defines a conjunction of equality constraints. A solver for the conjunctive fragment of the theory can be then used to determine if the conjunction is satisfiable. If the conjunction defined by a specific satisfying assignment π to $\mathsf{BoolSkel}(\varphi)$ is not satisfiable, the solver can *learn* $\neg\pi$ and iterate the process above with $\mathsf{BoolSkel}(\varphi) \wedge \neg\pi$. Propositional and theory reasoning alternate in this manner until a first-order structure satisfying the theory formula is found, or until the formula is shown to be unsatisfiable.

The primary aim of this paper is to explain and analyse DPLL(T) in the abstract interpretation framework. We show that reasoning about the Boolean structure and about conjunctions of theory facts is, in a strict, mathematical sense, an abstract interpretation of the semantics of a formula. Extensions of DPLL(T) such as theory propagation, early pruning, theory explanations, conflict set generation and generation of multiple reasons for a single conflict have natural characterisations in the language of abstract interpretation.

We emphasise that the purpose of this work is not to trivialise DPLL(T) by claiming it is "just abstract interpretation". Instead we aim to illuminate the link between SMT solvers and abstract interpretation to allow the transfer of results and intuition. Though some of our results are intuitively clear and known to the satisfiability community, our formalisation is not obvious. Our work shows that DPLL(T) is an instance of a generic, greatest fixed point computation that overapproximates the reduced product of two abstract domains. This result allows the static analysis community to better place DPLL(T) in the rich landscape of results concerning fixed point computations and domain combinations.

The secondary aim of this paper is to show that the product construction involved in DPLL(T) is sometimes unnecessary. We empirically compare splitting-on-demand [2], an extension of classic DPLL(T), with ACDCL [14,8], an algebraic generalisation of CDCL that does not operate over a product.[1]

Contributions. This paper makes the following contributions.

1. A new understanding of DPLL(T) within the abstract interpretation framework. We show that DPLL(T) is an instance of a product construction over a Boolean abstraction and a conjunctive theory abstraction.
2. A view of DPLL(T) as an instance of a more abstract procedure which permits combination of reasoning engines beyond the classic Boolean-theory split.
3. A empirical demonstration that, under some circumstances, the construction of products in DPLL(T) is unnecessary and detrimental to performance.

Related Work. A number of recent publications have given abstract interpretation accounts of decision procedures: [7] gives an account of propositional SAT procedures such as DPLL and CDCL using the same framework as this paper which

[1] Our benchmarks and an extended version of this paper with proofs can be found at http://www.cprover.org/papers/vmcai2013/

is the basis for the generalisation of CDCL in [8]. Independently of the above, [23] gives an abstract-interpretation account and generalisation of Stålmarck's method. In [6], Nelson-Oppen theory combination is characterised as a product construction over abstract domains.

A number of practical approaches have been derived directly from this point of view. These include extensions of the CDCL algorithm to the interval abstraction to decide floating-point logic [14] and reachability queries [9], and the synthesis of abstract transformers using the generalisation of Stålmarck's method mentioned above [22]. Before these, [15] proposed combining propositional SAT solvers and abstract interpreters in a DPLL(T)-style architecture.

A popular operational formalisation of DPLL(T) is given in [21]. Our work is closely related to research efforts to develop alternatives to DPLL(T). These approaches, called *natural-domain* SMT [4], lift the CDCL algorithm to operate directly on theory formulae. Notable examples have been presented for equality logic with uninterpreted functions [1], linear real arithmetic and difference logic [19,4], linear integer arithmetic [17], non-linear arithmetic [11,18], and floating-point arithmetic [14].

2 Abstract Satisfaction

This section provides a concise review of SMT [3], abstract interpretation [5], and the application of abstract interpretation to logic [7].

2.1 Satisfiability Modulo Theories

A *signature* Σ is a set of *function symbols* and *predicate symbols*, each associated with a non-negative *arity*. Predicate and function symbols with arity zero are called, respectively, *propositions* and *constants*. Ground terms are constants or function applications $f(t_1, \ldots, t_n)$ where f is an n-ary function and the t_i are ground terms. All formulae we consider are quantifier-free and have no first-order variables. For convenience, we omit these qualifiers in the rest of the paper. As is common in the SMT literature, we refer to uninterpreted constants as variables.

An *atomic formula* is a proposition, an n-ary predicate $p(t_1, \ldots, t_n)$ applied to terms $t_1, \ldots, t_n$, or a truth value in $\mathbb{B} = \{\mathsf{t}, \mathsf{f}\}$. A *literal* is an atomic formula or its negation. A literal is in *positive phase* if it is an atomic formula and in *negative phase* otherwise. For a literal l, we denote by $\mathsf{neg}(l)$ its opposite-phase counterpart. For a set of formulae Ψ we denote by $\neg\Psi$ the set $\{\neg\psi \mid \psi \in \Psi\}$. A *clause* is a disjunction of literals, and a *formula* is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses. We follow standard convention and denote clauses and CNF formulae as sets of literals, resp., sets of clauses where convenient. Unless otherwise specified, we assume all formulae to be in CNF. We denote by $\mathcal{A}(\varphi)$ the set of atomic subformulae of φ, by $\mathcal{L}(\varphi)$ the set of literals $\mathcal{A}(\varphi) \cup \neg\mathcal{A}(\varphi)$ and by $\mathcal{H}(\varphi)$ the set of terms occurring in φ. We denote by $\mathcal{V}(\varphi)$ the set of variables (uninterpreted constants) in φ.

Semantics. Formulae are interpreted over first-order structures. A *structure* for a signature Σ is a pair (U, ϵ) consisting of a non-empty set U called the *universe* and an *interpretation function* ϵ which maps every element of the signature to an appropriate object over U, e.g. constants are mapped to elements of U, n-ary functions to n-ary functions over U, etc. We denote (U, ϵ) simply by ϵ when U is clear from context or irrelevant. The *semantic entailment relation* $\models$ is defined as usual. Given a structure σ and formula φ, if $\sigma \models \varphi$ holds, then σ satisfies φ, and it is a *model* of φ. Otherwise, it is a *countermodel*.

Theories. We define a (Σ-)*theory* T_Σ as a set of first-order structures over a signature Σ (as is common in the SMT literature, e.g. [3]). We call a model $\sigma \in T_\Sigma$ of φ a T_Σ-model and a formula φ T_Σ-satisfiable if it has a T_Σ-model. The satisfiability problem modulo a theory T_Σ, for a quantifier-free ground formula φ, is to decide whether φ has a T_Σ-model.

Let P be a fixed set of propositions. A *propositional formula* is a P-formula, and a *propositional structure* or *propositional assignment* is an element of the set $\mathsf{PA}_P \mathrel{\hat{=}} P \to \mathbb{B}$. When discussing theories T_Σ in the context of propositional logic, we assume that P is disjoint from the signature Σ.

2.2 Abstract Interpretation

We briefly review some concepts in abstract interpretation. For convenience, we work in the Galois connection framework. We write $(C, \preceq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ for a Galois connection between the complete lattices C and A. An underapproximation is defined by a Galois connection $(C, \succeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsupseteq)$. In this paper, we assume all Galois connections we consider to satisfy $\gamma(\bot) = \bot$ and $\gamma(\top) = \top$. A *transformer* is a monotone function on a lattice. A transformer f on a complete lattice has a greatest fixed point, denoted $\mathsf{gfp}\ f$ or $\mathsf{gfp}\ X.\ f(X)$ and a least fixed point $\mathsf{lfp}\ f$ or $\mathsf{lfp}\ X.\ f(X)$. The gfp *closure* f^* of a transformer f is the transformer $a \mapsto \mathsf{gfp}\ X.\ f(X) \sqcap a$. The *best approximation* of $f : C \to C$ is $g \mathrel{\hat{=}} \alpha \circ f \circ \gamma$.

A *reduction operator* is a transformer ρ in an abstract domain A that is (i) *reductive*, i.e., for all $a \in A$ it holds that $\rho(a) \sqsubseteq a$ and (ii) *sound*, i.e., $\gamma \circ \rho = \gamma$. Reductions refine the representation of an abstract object without changing its meaning. A *dual reduction operator* generalises the representation without changing the meaning of an object.

Let $(A, \sqsubseteq)$ be an overapproximation of a powerset domain $(\wp(S), \subseteq)$ with $\wp(S) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. The *downset completion* of A is the lattice $\mathcal{D}(A) \mathrel{\hat{=}} (\mathsf{ds}(A), \subseteq)$ where $\mathsf{ds}(A)$ is the set of all $Q \in \wp(A)$ s.t. Q is *downwards closed*, i.e. $\forall a \in Q, a' \in A.\ a' \sqsubseteq a \implies a' \in Q$. When possible, we represent a set in $\mathsf{ds}(A)$ as the set of its maximal elements. It underapproximates the concrete domain $\wp(S)$ with $\alpha_\mathcal{D} : \wp(S) \to \mathcal{D}(A)$, $\alpha_\mathcal{D}(Q) \mathrel{\hat{=}} \{a \in A \mid \gamma(a) \subseteq Q\}$ and $\gamma_\mathcal{D} : \mathcal{D}(A) \to \wp(S)$, $\gamma_\mathcal{D}(D) \mathrel{\hat{=}} \bigcup_{d \in D} \gamma(d)$.

Let $(A, \sqsubseteq_A), (B, \sqsubseteq_B)$ be abstract domains over the concrete domain $(C, \subseteq)$, with Galois connections (α_A, γ_A) and (α_B, γ_B), respectively. The *Cartesian product* $A \times B$ is defined the abstract domain over the lattice $(A \times B, \sqsubseteq)$ with

$(a, b) \sqsubseteq (a', b')$ exactly if $a \sqsubseteq a'$ and $b \sqsubseteq b'$. There is a Galois connection to the concrete given by $\alpha_{A\times B}(c) = (\alpha_A(c), \alpha_B(c))$ and $\gamma_{A\times B}(a, b) = \gamma_A(a) \cap \gamma_B(b)$.

2.3 Interpreting Logics over Theories

We can use the formal machinery of abstract interpretation to approximate the meaning of logical formulae. Let T_Σ be a Σ-theory. The *concrete theory domain* of T_Σ is the powerset lattice $(\wp(T_\Sigma), \subseteq)$ together with the *model transformer* and *universal countermodel transformer* for each Σ-formula φ, given below.

$$mods_\varphi^{T_\Sigma}(S) \hat{=} \{\sigma \in T_\Sigma \mid \sigma \in S \wedge \sigma \models \varphi\}$$
$$ucmods_\varphi^{T_\Sigma}(S) \hat{=} \{\sigma \in T_\Sigma \mid \sigma \in S \vee \sigma \not\models \varphi\}$$

Abstractions of these operators are implemented in existing abstract domains for program analysis for the following reason. The function $mods_\varphi^{T_\Sigma}$ is equivalent to the strongest post-condition of an $\mathsf{assume}(\varphi)$ statement, while $ucmods_\varphi^{T_\Sigma}$ is equivalent to the weakest liberal pre-condition. In logical inference terms, $mods_\varphi^{T_\Sigma}$ implements *deduction*, since it maps a set of structures S to the strongest consequence of S w.r.t. φ, expressed as a set. Similarly, $ucmods_\varphi^{T_\Sigma}$ implements *abduction*, because it maps an element R to the weakest explanation for R.

Abstract domains and transformers can be used to perform sound but incomplete satisfiability checks. We refer to an abstraction of the concrete theory domain as an *abstract theory domain*.

Theorem 1 (Abstract Satisfaction). *Let amods be an overapproximation of* $mods_\varphi^{T_\Sigma}$ *and aucmods be an underapproximation of* $ucmods_\varphi^{T_\Sigma}$*. The formula* φ *is not* T_Σ*-satisfiable (i)* gfp *amods* $= \bot$ *or (ii)* lfp *aucmods* $= \top$*.*

Refutational Completeness in Abstract Interpretation. Let f be a concrete transformer and g be a sound approximation of f in a lattice A, and a be an element of A. Then g is γ*-complete at* a if $\gamma \circ g(a) = f \circ \gamma(a)$ holds.

We now introduce new notions of completeness to express adequate precision. The transformer g is $\gamma_\bot$*-complete* at $a \in A$ if $\gamma \circ g(a) = \bot$ exactly if $f \circ \gamma(a) = \bot$, and it is $\bot$*-complete* at $a \in A$ if $g(a) = \bot$ whenever $f \circ \gamma(a) = \bot$. If a transformer is $\bot$-complete at every element we simply say it is $\bot$-complete. The same holds for γ- and $\gamma_\bot$-completeness. A reduction operator is $\bot$-complete (respectively γ- or $\gamma_\bot$-complete) if it is complete w.r.t. the concrete identity function.

3 Boolean Reasoning as Abstract Interpretation

This section shows that the Boolean reasoning employed by the DPLL(T) algorithm is an instance of abstract interpretation. More precisely, we show that computing propositional solutions over the Boolean skeleton of a formula is an abstract interpretation of the formula's theory semantics.

Fix φ to be a Σ-formula and $P \subseteq \mathit{Props}$ to be a fresh set of propositions disjoint from Σ. We assume a bijective function $\mathsf{pmap} : \mathcal{A}(\varphi) \to P$ that relates the atoms in φ to the propositions in P.

Definition 1. *The* Boolean skeleton $\mathsf{BoolSkel}(\varphi)$ *is the propositional formula obtained by replacing each atomic formula* $\psi_{\mathcal{A}}$ *occurring in* φ *with* $\mathsf{pmap}(\psi_{\mathcal{A}})$.

Reasoning about Boolean structure can be understood as an abstraction of the semantics of a formula. From this perspective, the introduction of propositions for subformulae, and consequently the construction of an independent, propositional formula can be considered an implementation detail.

Definition 2. *For a set of* Σ*-formulae* F *we define the* Boolean abstraction Bool_F *as the abstract lattice* $(\wp(F \to \mathbb{B}), \subseteq)$ *with the Galois connection below.*

$$(\wp(T_\Sigma), \subseteq) \underset{\alpha_\mathsf{B}}{\overset{\gamma_\mathsf{B}}{\leftrightarrows}} (\mathsf{Bool}_F, \subseteq)$$

$$\alpha_\mathsf{B}(S) \mathrel{\hat{=}} \{\beta \in F \to \mathbb{B} \mid \exists \sigma \in S\ \forall \psi \in F.\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}\}$$

$$\gamma_\mathsf{B}(B) \mathrel{\hat{=}} \{\sigma \in T_\Sigma \mid \exists \beta \in B\ \forall \psi \in F.\ \sigma \models \psi \iff \beta(\psi) = \mathsf{t}\}$$

DPLL(T) applied to a formula φ employs the Boolean abstraction $\mathsf{Bool}_{\mathcal{A}(\varphi)}$. A set of propositional assignments from PA_P represents an element of $\mathsf{Bool}_{\mathcal{A}(\varphi)}$. We can move between these views by lifting pmap to map a set $S \subseteq \mathcal{A}(\varphi) \to \mathbb{B}$ bijectively to a subset of PA_P by mapping each assignment from subformulae to truth values to its corresponding assignment from propositions to truth values. Formally, we define $\mathsf{pmap}(S) \mathrel{\hat{=}} \{\lambda a.\beta(\mathsf{pmap}(a)) \mid \beta \in S\}$.

Relating Boolean Abstractions and the Skeleton. The set of propositional models can be computed by implementing an abstract transformer on $\mathsf{Bool}_{\mathcal{A}(\varphi)}$.

Proposition 1. *Let* $\psi = \mathsf{BoolSkel}(\varphi)$*, then the* skeleton transformer

$$\mathsf{BSkelModels} \mathrel{\hat{=}} \mathsf{pmap}^{-1} \circ \mathit{mods}_\psi^{\mathsf{PA}_P} \circ \mathsf{pmap}$$

is a sound overapproximation of the model transformer $\mathit{mods}_\varphi^{T_\Sigma}$.

The object amods_φ defined above is not the best overapproximation of the model transformer, since it only captures Boolean, but not theory reasoning. It is still precise when considered in the concrete.

Proposition 2. $\mathsf{BSkelModels}$ *is* $\gamma_\perp$*-complete w.r.t.* $\mathit{mods}_\varphi^{T_\Sigma}$.

In other words, even though the resulting element may not be the best abstract representation of the set of models of φ, its concretisation is precise. The remaining question is how one can determine whether the set of models it represents is empty. In DPLL(T), this is performed using a satisfiability check.

Definition 3. *The function* $\mathsf{BoolCheck} : \mathsf{Bool}_F \to \mathsf{Bool}_F$*, defined below, eliminates assignments not consistent in the theory.*

$$\mathsf{BoolCheck}(B) \mathrel{\hat{=}} \left\{\beta \in B \mid \bigwedge \{\varphi \mid \beta(\varphi) = \mathsf{t}\} \cup \{\neg\varphi \mid \beta(\varphi) = \mathsf{f}\} \text{ is } T_\Sigma\text{-}\mathsf{SAT}\right\}$$

Proposition 3. $\mathsf{BoolCheck}$ *is a* $\perp$*-complete reduction operator over* Bool_F.

Example 1. Consider the first-order formula below.

$$\varphi \;\hat{=}\; (x = y) \;\wedge\; (\neg(y = z) \;\vee\; \neg(x = z))$$

We fix the theory T to give equality its natural interpretation. We denote by $v_1v_2v_3$ the assignment $\{(x = y) \mapsto v_1, (y = z) \mapsto v_2, (x = z) \mapsto v_3\}$ in $\mathcal{A}(\varphi) \to \mathbb{B}$. For the mapping $\mathsf{pmap} \hat{=} \{(x = y) \mapsto p, (y = z) \mapsto q, (x = z) \mapsto r\}$ we obtain the Boolean skeleton below, which yields a skeleton transformer.

$$\mathsf{BoolSkel}(\varphi) \hat{=} p \wedge (\neg q \vee \neg r) \qquad \mathsf{BSkelModels}(\top) = \{\mathsf{ttf}, \mathsf{tft}, \mathsf{tff}\}$$

$\mathsf{BSkelModels}(\top)$ contains the assignment ttf which represents the empty set, since no structure in the theory satisfies $x = y$, $y = z$ but not $x = z$. The same holds for tft. Since both represent the empty set, this does not affect the precision of the transformer in the concrete, i.e., the transformer is γ-complete at $\top$ since $\gamma_{\mathsf{B}}(\mathsf{BSkelModels}(\top))$ is equal to $mods^{T}_{\varphi}(\top)$. Calling $\mathsf{BoolCheck}(\{\mathsf{ttf}, \mathsf{tft}, \mathsf{tff}\})$ refines the representation to $\{\mathsf{tff}\}$.

Satisfiability via Deduction and Reduction. We reformulate the initial step of DPLL(T) using abstract interpretation. Let $amods_{\varphi}$ be a $\gamma_{\perp}$-complete approximation of $mods^{T_\Sigma}_{\varphi}$ and let ρ be a $\perp$-complete reduction operator.

Step 1. Compute $a = amods_{\varphi}$ (e.g. with $amods_{\varphi} = \mathsf{BSkelModels}$)
Step 2. Return SAT if $\rho(a) \neq \perp$ (e.g. with $\rho = \mathsf{BoolCheck}$)

We can sketch DPLL(T) as depth-first variant of the above framework. Propositional models are enumerated on-the-fly by a SAT solver rather than computed in a single step; the reduction to $\perp$ is computed and checked by a theory solver. The following summarises the soundness and completeness argument.

Proposition 4. *If $amods_{\varphi}$ is $\gamma_{\perp}$-complete w.r.t. $mods^{T_\Sigma}_{\varphi}$ and ρ is a $\perp$-complete reduction, then $\rho(amods_{\varphi}(\top)) \neq \perp$ exactly if φ is T_Σ-satisfiable.*

3.1 Efficient Disjunction via the Cartesian Abstraction

The transformer $\mathsf{BSkelModels}$ generates the set of models of a propositional formula and is hence expensive to compute. Therefore, DPLL(T) instead uses a guided search process to enumerate models.

Partial Assignments and the Cartesian Abstraction. The main data structure for the guided search in a DPLL(T) solver is a partial assignment, a map from propositions to t, f, an unknown value $\top$ or a value $\perp$ representing a conflict. Partial assignments are refined using deduction and search. A partial assignment $f : P \to \{\mathsf{t}, \mathsf{f}, \top, \perp\}$ represents a set of propositional literals Q such that $f(p) = \mathsf{t}$, $f(p) = \mathsf{f}$, $f(p) = \top$ and $f(p) = \perp$ represent, respectively, that $p \in Q$, $\neg p \in Q$, $p, \neg p \notin Q$ and $p, \neg p \in Q$. Since we view the Boolean skeleton as an implementation detail, the description below directly uses atomic formulae.

Definition 4. *For a set of Σ-formulae F we define the* Cartesian abstraction Cart_F *as the abstract lattice $(\wp(F \cup \neg F), \sqsubseteq)$ with $\sqsubseteq = \supseteq$, $\bigsqcap = \bigcup$ and $\bigsqcup = \bigcap$. Cart_F abstracts Bool_F (and, as a consequence, the concrete theory domain). The Galois connections are as below.*

$$(\wp(T_\Sigma), \subseteq) \quad \underset{\alpha_\mathsf{B}}{\overset{\gamma_\mathsf{B}}{\leftrightarrows}} (\mathsf{Bool}_F, \subseteq) \underset{\alpha_\mathsf{BC}}{\overset{\gamma_\mathsf{BC}}{\leftrightarrows}} \quad (\mathsf{Cart}_F, \sqsubseteq)$$

$$\gamma_\mathsf{C} \;\hat{=}\; \gamma_\mathsf{BC} \circ \gamma_\mathsf{B} \qquad \alpha_\mathsf{C} \;\hat{=}\; \alpha_\mathsf{BC} \circ \alpha_\mathsf{B}$$

$$\alpha_\mathsf{BC}(B) \hat{=} \{\psi \mid \forall \beta \in B.\ \beta(\psi) = \mathsf{t}\} \sqcap \{\neg\psi \mid \forall \beta \in B.\ \beta(\psi) = \mathsf{f}\}$$

$$\gamma_\mathsf{BC}(\Theta) \hat{=} \{\beta \mid \forall \psi \in F.\ (\beta(\psi) = \mathsf{t} \Rightarrow \neg\psi \notin \theta) \wedge (\beta(\psi) = \mathsf{f} \Rightarrow \psi \notin \theta)\}$$

The use of propositional partial assignments in existing DPLL(T) solvers can be viewed as a way of representing $\mathsf{Cart}_{\mathcal{A}(\varphi)}$.

Unit Rule and BCP. DPLL(T) solvers perform Boolean reasoning over partial assignments using the *unit rule*, which states that if all but one literal in a propositional clause are contradicted by the current partial assignment, the remaining literal must be true. Below, we give the corresponding transformer over Cart_F.

Definition 5. *For a Σ-clause C and set of formulae F with $\mathcal{A}(C) \subseteq F$, the* unit rule *over Cart_F is the function $\mathsf{unit}_C^F : \mathsf{Cart}_F \to \mathsf{Cart}_F$ defined as:*

$$\mathsf{unit}_C^F(\Theta) \hat{=} \begin{cases} \bot & \text{if } \psi, \neg\psi \in \Theta \text{ or for all } l \in C,\ \mathsf{neg}(l) \in \Theta \\ \Theta \sqcap \{l\} & \text{else if } C = C' \cup \{l\} \text{ s.t. for all } l' \in C',\ \mathsf{neg}(l') \in \Theta \\ \Theta & \text{otherwise} \end{cases}$$

For a set of propositions P and propositional clause C, the propositional unit rule *is the rule $\mathsf{unit}_C^P : \mathsf{Cart}_P \to \mathsf{Cart}_P$.*

Example 2. Consider the formula from before, $\varphi \hat{=} (x = y) \wedge C$ where $C = (\neg(y = z) \vee \neg(x = z))$. We can apply $\mathsf{unit}_{x=y}^{\mathcal{A}(\varphi)}(\top)$ to obtain the element $\Theta = \{x = y\}$. Applying $\mathsf{unit}_C^{\mathcal{A}(\varphi)}(\Theta)$ gives no new information but simply returns Θ. We can refine the element with an unsound assumption by computing $\Theta' = \Theta \sqcap \{y = z\} = \{x = y, y = z\}$. Now, applying $\mathsf{unit}_C^{\mathcal{A}(\varphi)}(\Theta')$ yields $\Theta' \sqcap \{\neg(x = z)\}$.

Unit rule applications soundly approximate the model transformer, regardless of the underlying theory.

Proposition 5. *Let C be a clause such that $\mathcal{A}(C) \subseteq F$. For any theory T_Σ, the transformer unit_C^F is a sound approximation of $\mathsf{mods}_C^{T_\Sigma}$.*

DPLL(T) solvers use a process called Boolean Constraint Propagation (BCP) in which the unit rule is applied exhaustively to deduce new theory facts. This process computes a greatest fixed point with the function defined earlier.

Definition 6. *For a Σ-formula φ and a set of Σ-formulae $F \supseteq \mathcal{A}(\varphi)$, the* BCP *transformer $\mathsf{bcp}_\varphi : \mathsf{Cart}_F \to \mathsf{Cart}_F$ is the following function.*

$$\mathsf{bcp}_\varphi(\Theta) \hat{=} \mathsf{gfp}\ X.\ \bigsqcap_{C \in \varphi} \mathsf{unit}_C^F(X \sqcap \Theta)$$

During the run of DPLL(T), the propositional formula changes in a process called learning. Here, we take the point of view that the use of a propositional formula is an implementation detail. Changing the propositional formula then amounts to refining the model transformer over the Cartesian abstraction.

3.2 Satisfiability via Abstract Splitting

In lazy DPLL(T), theory consistency is checked once a partial assignment that satisfies every clause is found. The following operator is used for the check.

Definition 7. *We define* $\mathsf{CartCheck} : \mathsf{Cart}_F \to \mathsf{Cart}_F$ *as*

$$\mathsf{CartCheck}(\Theta) \hat{=} \begin{cases} \bot & \text{if } \bigwedge \theta \text{ is not } T_\Sigma\text{-satisfiable} \\ \Theta & \text{otherwise} \end{cases}$$

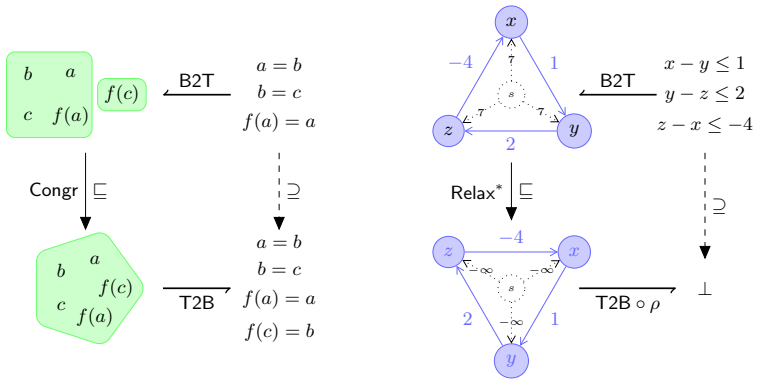
Proposition 6. $\mathsf{CartCheck}$ *is a* $\bot$*-complete reduction operator.*

The previous section showed that bcp_φ soundly approximates the model transformer and $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ is a $\bot$-complete reduction. Proposition 4 cannot be applied though, since bcp_φ lacks the necessary completeness requirement and solely performing deduction and reduction does not give a complete procedure. In the absence of this global completeness, DPLL(T) searches for points at which the model transformer is locally complete. The proposition below shows that a common stopping criterion in DPLL(T) is a local completeness check.

Proposition 7. *Let* φ *be a* Σ*-formula in* CNF*, and let* $\Theta \in \mathsf{Cart}_{\mathcal{A}(\varphi)}$ *such that for every clause* $C \in \varphi$ *there is a literal* $l \in C$ *such that* $l \in \Theta$*. Then* bcp_φ *is* γ*-complete at* Θ*.*

The search proceeds as follows. After the BCP step, classic DPLL chooses a variable in a partial assignment that is assigned to $\top$ and explores separately the cases where it is t and f. In terms of abstract interpretation this amounts to decomposing a partial assignment a into two more precise assignments a_1, a_2 that, taken together, have the same meaning as the original assignment, i.e., $\gamma(a_1) \cup \gamma(a_2) = a$. Let $amods_\varphi : A \to A$ be a sound approximation of $mods_\varphi^{T_\Sigma}$ and let $\rho : A \to A$ be a $\bot$-complete reduction, then we can state the abstract algorithm as follows.

(Init). Let $a_{\mathsf{init}} = \top$.
Step 1. Compute the greatest fixed point $a = \mathsf{gfp}\ X.amods_\varphi(X \sqcap a_{\mathsf{init}})$.
Step 2. If $a = \bot$ then return.
Step 3. If $amods_\varphi$ is $\gamma_\bot$-complete at a and $\rho(a) \neq \bot$ then return SAT.
Step 4. Split a into two smaller elements a_1, a_2 s.t. $a_1 \sqsubset a$, $a_2 \sqsubset a$ and $\gamma(a_1) \cup \gamma(a_2) = \gamma(a)$, and call the algorithm recursively.
(a) If a call with $a_{\mathsf{init}} = a_1$ returns SAT then return SAT
(b) If a call with $a_{\mathsf{init}} = a_2$ returns SAT then return SAT



(a) EUF with congruence closure (b) DL with Bellman-Ford

Fig. 1. Examples of theory solvers as abstract domains

Example 3. Consider again the formula $\varphi \mathrel{\hat{=}} x = y \wedge C$ where C is the clause $(\neg(y = z) \vee \neg(x = z))$ We fix the theory T to give equality its natural interpretation. Computing $\mathsf{bcp}_\varphi(\top)$ yields the result $a = \{x = y\}$. This is not γ-complete reasoning, since it abstracts structures where $x = y = z$, which are not models. We split Θ into the smaller elements $a_1 = a \sqcap \{y = z\}$ and $a_2 = a \sqcap \{\neg(y = z)\}$. In the first recursive call, we obtain $a' = \{x = y, y = z, \neg(x = z)\}$ from $\mathsf{bcp}_\varphi(a_1)$. The transformer bcp_φ is γ-complete at a', therefore we know that a' is a set of models. It remains to check whether a' is an empty set of models, by calling $\mathsf{CartCheck}(a')$, which returns $\bot$. In the second recursive call, BCP yields no further refinement. But bcp_φ is already γ-complete at a_2, therefore we check the conjunction $(x = y) \wedge \neg(y = z)$ with $\mathsf{CartCheck}(a_2)$. The check returns a_2, indicating that a_2 represents a non-empty set and we return SAT.

Depending on details of the logic and abstract domain used the above algorithm may not be complete, i.e., it may not return SAT exactly if φ is satisfiable. We will discuss conditions for completeness in a bit more detail later. Whenever the algorithm returns SAT, then the formula is satisfiable.

Proposition 8. *Let $\mathit{amods}_\varphi : A \to A$ be an overapproximation of $\mathit{mods}_\varphi^{T_\Sigma}$ and ρ be a $\bot$-complete reduction operator. If for some element $a \in A$, amods_φ is $\gamma_\bot$-complete at a and $\rho(\mathit{amods}_\varphi(a)) \neq \bot$, then φ is satisfiable.*

4 Theory Solvers as Abstract Domains

In this section, we show that theory solvers for equality with uninterpreted functions, and for difference logic can be viewed as reduction operators. These serve as examples of the general approach as it is not feasible to cover all theory solvers in one paper.

4.1 Equality with Uninterpreted Functions

An *equality formula* contains the predicate $=$ and function symbols. We use $t \neq t'$ as a shorthand to denote $\neg(t = t')$. We define the theory of Equality with Uninterpreted Functions (EUF) as the set T_{EUF} containing all structures $(\mathbb{Z}, \epsilon)$ where ϵ interprets $=$ as the standard equality relation over $\mathbb{Z}$. The congruence closure algorithm decides satisfiability of conjunctions of equality literals. The algorithm constructs congruence classes containing terms from $\mathcal{H}(\varphi)$ (often implemented using union-find data structures) and a set of pairs in $\mathcal{H}(\varphi)$ that are known to be unequal. The data structure used by congruence closure forms a lattice. A *partition* of a set X is a collection of disjoint, non-empty subsets of X whose union is X. $\mathsf{Part}(X)$ denotes the partitions of a set X.

Definition 8. *For an* EUF φ*, the* EUF abstraction*,* EUF_φ *is* $(\mathsf{TS}, \sqsubseteq)$ *where:*

$$\mathsf{TS} \mathrel{\hat{=}} \mathsf{Part}(\mathcal{H}(\varphi)) \times \wp(\mathcal{H}(\varphi) \times \mathcal{H}(\varphi))$$

and $(P, D) \sqsubseteq (P', D')$ *exactly if* $\forall p' \in P'. \exists p \in P$ *s.t.* $p \supseteq p'$ *and* $D \supseteq D'$*. Note that* EUF_φ *abstracts the concrete and refines* $\mathsf{Cart}_{\mathcal{A}(\varphi)}$*. As both domains are lattices,* α_{TS} *and* $\mathsf{B2T}$ *are uniquely defined from* γ_{TS} *and* $\mathsf{T2B}$*.*

$$(\wp(T_\Sigma), \subseteq) \underset{\alpha_{\mathsf{TS}}}{\overset{\gamma_{\mathsf{TS}}}{\leftrightarrows}} (\mathsf{TS}, \sqsubseteq) \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} (\mathsf{Cart}_{\mathcal{A}(\varphi)}, \supseteq)$$

$$\gamma_{\mathsf{TS}}(P, D) \mathrel{\hat{=}} \{\sigma \mid \forall (t_1, t_2) \in D.\ \sigma \models t_1 \neq t_2 \wedge \forall p \in P\ \forall t_1, t_2 \in p.\ \sigma \models t_1 = t_2\}$$

$$\mathsf{T2B}(P, D) \mathrel{\hat{=}} \mathcal{L}(\varphi) \cap (\{t_1 = t_2 \mid \exists p \in P.\ t_1, t_2 \in p\} \cup \{t_1 \neq t_2 \mid (t_1, t_2) \in D\})$$

We define the steps of the algorithm as transformers over the abstraction. A *congruence operator* $\mathsf{Congr} : \mathsf{EUF}_\varphi \to \mathsf{EUF}_\varphi$ merges the congruence classes of two terms if all their subterms s, t are pairwise *congruent* in the current element P, i.e., if they are in the same congruence class. If in $(P, D) \in \mathsf{EUF}_\varphi$ terms are found to both equal and unequal, i.e., for some $p \in P$ and $(t_1, t_2) \in D$ it holds that $t_1, t_2 \in p$, then $\bot$ is returned. Otherwise, we define for a partition $P = \{p_1, \ldots, p_k\}$:

$$\mathsf{Congr}(P, D) \mathrel{\hat{=}} \begin{cases} (P \setminus \{p, p'\} \cup \{p \cup p'\}, D) \text{ for some disjoint } p, p' \in P \text{ s.t.} \\ \quad f(s_1, \ldots, s_k) \in p, f(t_1, \ldots, t_k) \in p' \text{ s.t. all } s_i, t_i \text{ are congr. in } P \\ (P, D) \text{ if no such } p, p' \text{ exist} \end{cases}$$

The congruence operator is reductive (it gains in precision in each step), and refines the representation of a set of structures without changing the set itself. Figure 1(a) illustrates Congr along with the Galois connection between the EUF and the Cartesian abstractions. The set of formulae in the top right can be concretised to the pair of congruence classes in the top left. These are then merged by Congr as $a = c$ implies $f(a) = f(c)$ and finally can be abstracted to give the set of formulae in the bottom right; simulating inference in the Cartesian domain.

Proposition 9. Congr *is a reduction operator.*

The congruence closure algorithm then computes the greatest fixed point gfp Congr over EUF_φ by iterating Congr until no new information can be deduced. It is a refutationally complete procedure, i.e., if a conjunction of equality literals is empty, then the fixed point will be $\bot$.

Proposition 10. *The* gfp *closure* Congr^* *is* $\bot$*-complete.*

4.2 Difference Logic

Formulae in *difference logic* (DL) contain the binary function symbol $-$ and the binary predicate $\leq$, and have atoms of the form $x - y \leq c$. The theory of *integer difference logic* (T_{IDL}) is the set of structures of the form $(\mathbb{Z}, \epsilon)$ where ϵ maps the symbols $\leq$ and $-$ to their natural interpretations over the integers.

A conjunct of difference logic atoms can be modelled by a weighted directed graph in which the set of nodes N corresponds to the set of variables in the conjunct. An atom $x - y \leq c$ is denoted as an edge (x, y) with weight c. The conjunct is satisfiable if and only if the graph contains no negative cycles.

Negative cycles can be detected using the Bellman-Ford algorithm (BF). The main data structure of BF associates a *weight* in $\mathbb{Z}_\infty \mathrel{\hat{=}} \mathbb{Z} \cup \{-\infty, \infty\}$ with each node n. The weight is an upper bound on the shortest path from the source to n. The weight $-\infty$ indicates a negative cycle. For handling DL, we choose the source to be s, a fresh node, and assume that s is connected to all variables with weight M_φ, which is an integer constant larger than the longest possible path.[2] The initial node weights are also M_φ. Node weights are reduced in each round if there is a neighbouring node that gives a shorter, negative cost path. After $|N|-1$ iterations, the path lengths will have converged if and only if there are no negative cycles. If a final iteration changes the scores, the graph contains a negative cycle.

We make two observations which allow us to simplify presentation: (i) since edge weights represent upper bounds on the minimal distance between two variables, node weights can simply be viewed as special edges (s, n), (ii) BF can then be viewed to operate solely over edge weights (missing edges are given weight ∞). For a formula φ, we define the edge set E_φ as the set $(\{s\} \cup \mathcal{V}(\varphi)) \times \mathcal{V}(\varphi)$, where s is the fresh source node.

Definition 9. *For a* DL*-formula* φ*, the* BF abstraction BF_φ *is* $(\mathsf{TS}, \sqsubseteq)$ *where:*

$$\mathsf{TS} \mathrel{\hat{=}} \{f : E_\varphi \to \mathbb{Z}_\infty \mid \forall x \in \mathcal{V}(\varphi).\ f(s, x) \leq M_\varphi\}$$
$$f \sqsubseteq g \text{ iff } \forall e \in E_\varphi.\ f(e) \leq g(e)$$

BF_φ *abstracts the concrete and refines* $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ *and again, only half of each Galois connection is explicitly defined.*

$$(\wp(T_\Sigma), \subseteq) \underset{\alpha_{\mathsf{TS}}}{\overset{\gamma_{\mathsf{TS}}}{\leftrightarrows}} (\mathsf{TS}, \sqsubseteq) \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} (\mathsf{Cart}_{\mathcal{A}(\varphi)}, \supseteq)$$
$$\gamma_{\mathsf{TS}}(f) \mathrel{\hat{=}} \{\sigma \mid \forall (x, y) \in \mathcal{V}(\varphi) \times \mathcal{V}(\varphi).\ \sigma \models\ x - y \leq f(x, y)\}$$
$$\mathsf{B2T}(\Theta) \mathrel{\hat{=}} \lambda(x, y).\ \min(\{k \mid x - y \leq k \in \Theta\} \cup \{\top_{\mathsf{BF}}(x, y)\})$$

[2] E.g., M_φ can be the sum of the absolute values of all the integer constants in φ.

As in the case of EUF, the steps of the algorithm are reduction operators. In the case of BF, there are two reductions; the relax step and the cycle check.

Proposition 11. $\mathsf{Relax} : \mathsf{BF}_\varphi \to \mathsf{BF}_\varphi$ *and* $\mathsf{NegC} : \mathsf{BF}_\varphi \to \mathsf{BF}_\varphi$ *are reductions:*

$$\mathsf{Relax}(f)(x,y) \mathrel{\hat{=}} \begin{cases} f(x,y) & x \neq s \\ \min(\{f(x,y)\} \cup \{f(x,z) + f(z,y) \mid z \in \mathcal{V}(\varphi)\}) & x = s \end{cases}$$

$$\mathsf{NegC}(f) \mathrel{\hat{=}} \begin{cases} \bot & \textit{if } \mathsf{Relax}^{|\mathcal{V}(\varphi)|} \neq \mathsf{Relax}^{|\mathcal{V}(\varphi)|-1} \\ \mathsf{Relax}^{|\mathcal{V}(\varphi)|} & \textit{otherwise} \end{cases}$$

In addition to the above function, consider a simple canonicity reduction ρ s.t. $\rho(f) = \bot$ if f maps some edge to $-\infty$ and $\rho(f) = f$ otherwise. Relax, ρ and the Galois connections to the Cartesian domain are shown in Figure 1(b). Similarly to Figure 1(a), the Cartesian domain is on the right and by mapping to the concrete (BF on the left) and performing reduction, it is possible to find the inconsistency. The function NegC can then be viewed as a fixed point computation (not based on Kleene iteration) over the relaxation function.

Proposition 12. NegC *computes the fixed point* $(\rho \circ \mathsf{Relax})^*$ *and is* $\bot$*-complete.*

5 DPLL(T) as a Product Construction

We have given separate accounts of the Boolean and theory reasoning components of DPLL(T) as abstract interpretation. We now show that DPLL(T) can be viewed to compute a fixed points over a product between the Cartesian abstraction over the formula atoms $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ and an abstract theory domain TS.

Definition 10. *We define a* DPLL(T) theory domain *to be an abstract lattice* $(\mathsf{TS}, \sqsubseteq)$ *such that the following conditions hold.*

(*i*) TS *abstracts the concrete with Galois connection* $(\alpha_{\mathsf{TS}}, \gamma_{\mathsf{TS}})$,
(*ii*) $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ *abstracts* TS *with Galois connection* $(\mathsf{T2B}, \mathsf{B2T})$,
(*iii*) $\gamma_{\mathsf{C}} = \gamma_{\mathsf{TS}} \circ \mathsf{B2T}$ *and* $\alpha_{\mathsf{C}} = \alpha_{\mathsf{TS}} \circ \mathsf{T2B}$.

$$(\wp(T_\Sigma), \subseteq) \quad \underset{\alpha_{\mathsf{TS}}}{\overset{\gamma_{\mathsf{TS}}}{\leftrightarrows}} (\mathsf{TS}, \sqsubseteq) \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} \quad \gamma_{\mathsf{C}} \mathrel{\hat{=}} \gamma_{\mathsf{TS}} \circ \mathsf{B2T}, \; \alpha_{\mathsf{C}} \mathrel{\hat{=}} \mathsf{T2B} \circ \alpha_{\mathsf{TS}} \quad (\mathsf{Cart}_{\mathcal{A}(\varphi)}, \sqsubseteq)$$

The first condition ensures that datastructure of the theory solver represent sets of T_Σ structures. The other conditions require some motivation: The second condition ensures that conjunctions of literals in $\mathcal{A}(\varphi)$ can be expressed in TS without loss of precision. This corresponds to the requirement that the logic fragment handled by the theory solver includes conjunctions over $\mathcal{A}(\varphi) \cup \neg\mathcal{A}(\varphi)$, i.e., that satisfiability queries generated by CartCheck can be expressed. For convenience, we use a Galois connection to model this relation, even though in practice a weaker relation between the two might suffice. We assume that T2B and B2T can be computed. The third condition ensures that the Galois connections are compatible. We can now formally define DPLL(T) abstractions.

Definition 11. *For a T_Σ-formula φ and a* DPLL(T) *theory domain* TS*, the* DPLL(T) abstract domain $\mathsf{DPLL}(\mathsf{TS})$ *is the product domain* $\mathsf{Cart}_{\mathcal{A}(\varphi)} \times \mathsf{TS}$.

Example 4. We consider equality formulae φ. EUF_φ is a DPLL(T) theory domain, since it abstracts the concrete, and it refines the Cartesian abstraction.

We illustrate operations described in this section over $\mathsf{DPLL}(\mathsf{EUF}_\varphi)$. For convenience, we denote for three terms x, $f(x)$, z the partition $\{\{x\}, \{f(x), z\}\}$ either by $[x][f(x), z]$ or simply by $[f(x), z]$, omitting singleton partitions.

BCP with Theory Propagation. The classic DPLL(T) architecture only uses theory reasoning to check satisfiability of candidates. *Theory propagation* is a common refinement of this basic architecture. There, an element $\Theta \in \mathsf{Cart}_{\mathcal{A}(\varphi)}$ is refined with information deduced in the theory solver. One propagation step in a DPLL(T) solver with theory propagation can be broken down into these substeps:

(i) *Boolean deduction:* Perform Boolean reasoning.
(ii) *Theory instantiation:* Communicate Boolean facts to theory.
(iii) *Theory deduction:* Perform theory reasoning.
(iv) *Theory propagation:* Find implied Boolean consequences.

Definition 12. *We define the* theory instantiation *and* theory propagation *transformers over* DPLL(T) *below.*

$$\mathsf{tinst}(\Theta, \mathsf{te}) \hat{=} (\Theta, \mathsf{te} \sqcap \mathsf{B2T}(\Theta)) \qquad \mathsf{tprop}(\Theta, \mathsf{te}) \hat{=} (\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te})$$

Example 5. We assume that $\mathcal{A}(\varphi) = \{x = y,\ y = z\}$. Consider the element $(\Theta, \mathsf{te}) \hat{=} (\{x = y\}, ([x][y][z], \{y, z\})$ of $\mathsf{DPLL}(\mathsf{EUF}_\varphi)$. Applying $\mathsf{tinst}(\Theta, \mathsf{te})$ yields $(\Theta, ([x, y][z], \{y, z\}))$. Applying $\mathsf{tprop}(\Theta, \mathsf{te})$ yields $(\{x = y, \neg(y = z)\}, \mathsf{te})$. Neither operator changes the semantics of the tuple.

Proposition 13. *The transformers* tinst *and* tprop *are reductions over* $\mathsf{DPLL}(\mathsf{TS})$.

We note that *early pruning* [3] is just a special case of theory propagation in the lattice theoretic setting, i.e., it is the case where theory propagation finds $\bot$.

Deduction over $\mathsf{Cart}_{\mathcal{A}(\varphi)}$ is performed using the unit rule, while deduction inside the theory solver is handled by some reduction operator.

Definition 13. *The* Boolean deduction transformer bded_φ *is a sound overapproximation of* $mods_\varphi^{T_\Sigma}$ *over* $\mathsf{Cart}_{\mathcal{A}(\varphi)}$.

In practice, $\mathsf{bded}_\varphi = \mathsf{bcp}_\varphi$, but in principle other sound abstract transformers could be used.

Definition 14. *A* theory deduction transformer tded *is a reduction over* TS.

We extend the functions bded_φ and tded to $\mathsf{DPLL}(\mathsf{TS})$ as follows.

$$\mathsf{bded}_\varphi^\times(\Theta, \mathsf{te}) \hat{=} (\mathsf{bded}_\varphi(\Theta), \mathsf{te}) \qquad \mathsf{tded}^\times(\Theta, \mathsf{te}) \hat{=} (\Theta, \mathsf{tded}(\mathsf{te}))$$

We can now describe BCP with theory deduction as the following function, which executes the steps listed in the beginning of this section.

Definition 15. *We define* $\mathsf{deduce}_\varphi : \mathsf{DPLL(TS)} \to \mathsf{DPLL(TS)}$ *as follows.*

$$\mathsf{deduce}_\varphi \mathrel{\hat{=}} \mathsf{tprop} \circ \mathsf{tded}^\times \circ \mathsf{tinst} \circ \mathsf{bded}^\times_\varphi$$

Proposition 14. deduce_φ *is a sound overapproximation of* $mods^{T_\Sigma}_\varphi$.

Example 6. Consider the formula φ given as $f(x) = y \wedge x = z \wedge (f(z) \neq y \vee y = z)$. We compute deduce_φ, starting from $(\top, \top)$. Applying $\mathsf{bded}^\times_\varphi(\top, \top)$ refines the left-hand side to $\{f(x) = y, x = z\}$. Applying tinst communicates the deduction to the theory and obtains $([f(x), y][x, z], \emptyset)$ on the right. Theory deduction tded refines this to $([f(x), y, f(z)][x, z], \emptyset)$ using congruence. Finally, theory propagation tprop obtains $\{f(x) = y, x = z, f(z) = y\}$ on the left.

The deduction step in DPLL(T) computes a greatest fixed point over deduce_φ. A decision over an element Θ constructs an assignment $\Theta \cup l$, where l is a literal that occurs in neither positive nor negative phase in Θ. In abstract-interpretation terminology, this corresponds to a jump down the lattice which underapproximates the greatest fixed point and can be viewed as a dual widening operator [7].

Conflict Analysis with Theory Explanations. DPLL(T) solvers are based on propositional clause learning algorithms. The power of these algorithms rests significantly in the conflict analysis step, which extracts general, sufficient conditions for unsatisfiability from specific contradictory cases. We describe conflict analysis abstractly (see [14,8] for a lifting of conflict analysis algorithms to abstract domains). Conflict analysis computes a least fixed point over sets of elements over the underlying domain [7]: In general, there may be incomparable reasons a and b for a given deduction c, the most general conflict analysis will therefore return the set $\{a, b\}$. Indeed, conflict analyses that collect more than one conflict do exist [16].

In order to integrate theory solvers meaningfully into the analysis, they need to be able to supply explanations for deduced facts whenever theory propagation was applied. A step during conflict analysis with theory explanations can be broken down into the following substeps.

(i) *Boolean abduction:* Find Boolean conflict explanations.
(ii) *Theory justification:* Delegate explanations to the theory solver.
(iii) *Theory abduction:* Find theory explanations.
(iv) *Theory explanation:* Translate theory explanation into Boolean facts.

Recall that deduction corresponds to overapproximation of $mods^{T_\Sigma}_\varphi$. Conversely, finding explanations for deductions corresponds to underapproximation of the $ucmods^{T_\Sigma}_\varphi$ transformer.

Definition 16. *A* Boolean abduction transformer babd_φ *is an underapproximation of* $ucmods^{T_\Sigma}_\varphi$ *over the downset completion* $\mathcal{D}(\mathsf{Cart}_{\mathcal{A}(\varphi)})$.

Example 7. Consider $\varphi \mathrel{\hat{=}} \varphi' \wedge (x \neq y \vee r = z) \wedge (x = y \vee r \neq z)$. Assume that the element $\Theta \mathrel{\hat{=}} \{x = y, r = z\}$ leads to a contradiction. A sound abduction may

obtain $\mathsf{babd}_\varphi(\{\Theta\}) = \{\{x = y\}, \{r = z\}\}$, indicating that $x = y$ and $r = z$ are both explanations for Θ, since one element in Θ suffices to deduce the other.

Theory solvers have no access to the original formula φ, but only to their internal state. Essentially, they correspond to abduction with respect to the truth-constant t.

Definition 17. *A* theory abduction transformer tabd *is a dual reduction over the downset completion* $\mathcal{D}(\mathsf{TS})$.

Example 8. Consider $\mathsf{te} = ([x, y, z], \{(x, y), (y, z)\})$, which represents a conflict. A sound abduction may return $\mathsf{tabd}(\{\mathsf{te}\}) = \{([x, y], \{(x, y)\})\ ,\ ([y, z], \{(y, z)\})\}$, highlighting two separate reasons for the conflict.

We extend the functions babd_φ and tabd to sets in $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ as follows.

$$\mathsf{babd}^\times_\varphi(\Gamma) \hat{=} \{(\Theta, \mathsf{te}) \mid \exists (\Theta', \mathsf{te}) \in \Gamma.\ \Theta \in \mathsf{babd}_\varphi(\{\Theta'\})\}$$
$$\mathsf{tabd}^\times(\Gamma) \hat{=} \{(\Theta, \mathsf{te}) \mid \exists (\Theta, \mathsf{te}') \in \Gamma.\ \mathsf{te} \in \mathsf{tabd}(\{\mathsf{te}'\})\}$$

The above transformers find reasons in their respective domains. The transformers we define next explain facts by crossing domain boundaries. When crossing from the theory abstraction to the less precise Cartesian abstraction the issue of expressibility arises, since some abstract theory facts may not have precise counterparts in the Cartesian domain. For an element $\mathsf{te} \in \mathsf{TS}$, we write *expressible*$(\mathsf{te})$ to denote the condition that te is precisely expressible in $\mathsf{Cart}_{\mathcal{A}(\varphi)}$, i.e. $\gamma_{\mathsf{TS}}(\mathsf{te}) = \gamma_{\mathsf{C}} \circ \mathsf{T2B}(\mathsf{te})$.

Definition 18. *We define the* theory justification *and* theory explanation *transformer over* $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ *below.*

$$\mathsf{tjustify}(\Gamma) \hat{=} \{(\Theta, \mathsf{B2T}(\Theta') \sqcap \mathsf{te}) \mid (\Theta \sqcap \Theta', \mathsf{te}) \in \Gamma\}$$
$$\mathsf{texpl}(\Gamma) \hat{=} \{(\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te}') \mid (\Theta, \mathsf{te} \sqcap \mathsf{te}') \in \Gamma \textit{ s.t. expressible}(\mathsf{te})\}$$

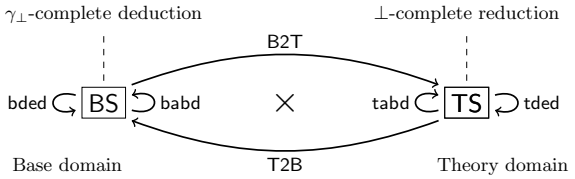
Example 9. Consider a set of atoms $\mathcal{A}(\varphi) = \{x = y,\ y = z\}$, and an element (θ, te) with $\theta = \{x = y\}$ and $\mathsf{te} = ([x][y][z], \{(y, z)\})$. Then $\mathsf{tjustify}(\{(\theta, \mathsf{te})\})$ contains the justification $(\top, ([x, y][z], \{(y, z)\})$, and $\mathsf{texpl}(\{(\theta, \mathsf{te})\})$ contains the explanation $(\{x = y,\ \neg(y = z)\}, \top)$.

The transformer $\mathsf{tjustify}$ explains information from the Cartesian domain in terms of the theory domain. The transformer texpl does the opposite, but can only do so if a given theory domain fact can be precisely expressed in $\mathsf{Cart}_{\mathcal{A}(\varphi)}$. In both cases, the formula φ is not taken into consideration.

Proposition 15. $\mathsf{tjustify}$ *and* texpl *are dual reductions.*

We note that *conflict set generation* [3] is a combination of theory abduction tabd of the $\bot$ element, followed by theory explanation.

A step of conflict analysis with theory justification can then be modelled as a function that executes the steps outlined in the beginning of this section.



	Model Search
Domain	$\mathsf{BS} \times \mathsf{TS}$ s.t. $\mathsf{TS} \underset{\mathsf{T2B}}{\overset{\mathsf{B2T}}{\leftrightarrows}} \mathsf{BS}$
Req. Transfomers	$\mathsf{bded} : \mathsf{BS} \to \mathsf{BS}$, $\mathsf{tded} : \mathsf{TS} \to \mathsf{TS}$ overapprox. of $mods_\varphi^{T_\Sigma}$
Theory Instantiation	$\mathsf{tinst}(\mathsf{be}, \mathsf{te}) \mathrel{\hat{=}} (\mathsf{be}, \mathsf{te} \sqcap \gamma_{\mathsf{TS}}(\mathsf{be}))$
Theory Propagation	$\mathsf{tprop}(\mathsf{be}, \mathsf{te}) \mathrel{\hat{=}} (\mathsf{be} \sqcap \alpha_{\mathsf{TS}}(\mathsf{te}), \mathsf{te})$
Deduction	$\mathsf{deduce}_\varphi \mathrel{\hat{=}} \mathsf{tprop} \circ \mathsf{tded}^\times \circ \mathsf{tinst} \circ \mathsf{bded}^\times$
Model Search	gfp over deduce_φ with dual widening over BS

	Conflict Analysis
Domain	Downset completion $\mathcal{D}(\mathsf{BS} \times \mathsf{TS})$
Req. Transfomers	babd over $\mathcal{D}(\mathsf{BS})$, tabd over $\mathcal{D}(\mathsf{TS})$ u.-approx. of $ucmods_\varphi^{T_\Sigma}$
Theory Justification	$\mathsf{tinst}(\Gamma) \mathrel{\hat{=}} \{(\Theta, \gamma_{\mathsf{TS}}(\Theta') \sqcap \mathsf{te}) \mid (\Theta \sqcap \Theta', \mathsf{te}) \in \Gamma\}$
Theory Explanation	$\mathsf{texpl}(\Gamma) \mathrel{\hat{=}} \{(\Theta \sqcap \mathsf{T2B}(\mathsf{te}), \mathsf{te}') \mid (\Theta, \mathsf{te} \sqcap \mathsf{te}') \in \Gamma$ s.t. $expressible(\mathsf{te})\}$
Abduction	$\mathsf{abduce}_\varphi(\Gamma) \mathrel{\hat{=}} \mathsf{texpl} \circ \mathsf{tabd}^\times \circ \mathsf{tjustify} \circ \mathsf{babd}^\times$
Conflict Analysis	lfp over abduce_φ with dual narrowing

Fig. 2. DPLL(T) as Abstraction

Definition 19. *We define the transformer* abduce_φ *over* $\mathcal{D}(\mathsf{DPLL}(\mathsf{TS}))$ *as:*

$$\mathsf{abduce}_\varphi \mathrel{\hat{=}} \mathsf{texpl} \circ \mathsf{tabd}^\times \circ \mathsf{tjustify} \circ \mathsf{babd}^\times_\varphi$$

Proposition 16. abduce_φ *is a sound underapproximation of* $ucmods_\varphi^{T_\Sigma}$.

Conflict analysis can then be viewed to compute a least fixed point over abduce_φ, starting from a propositional conflict $\{(\perp, \mathsf{te})\}$ or theory conflict $\{(\Theta, \perp)\}$. In practice, solvers do not keep track of sets of explanations for a conflict, but will instead consider only one. Choosing specific explanations can be viewed as a dual narrowing, since it underapproximates a least fixed point [7].

6 Algebraic Extensions of DPLL(T)

In this section, we first generalise the product construction of DPLL(T) and then show empirically that the communication restrictions induced by products are sometimes unnecessary and disadvantageous.

An Abstract View of DPLL(T). The overall architecture, domains and required transformers for DPLL(T) are depicted in Figure 2. We view the product construction $\mathsf{DPLL}(\mathsf{TS})$, as a special instance of a more general construction in which the Cartesian abstraction is a parameter. Due to space constraints, we only cover splitting-based DPLL(T) formally.

Definition 20. *An* abstract DPLL(T) domain *for a* base domain BS *and* theory domain TS *is the domain* $\mathsf{ADPLL}(\mathsf{BS},\mathsf{TS}) \mathrel{\hat{=}} \mathsf{BS} \times \mathsf{TS}$ *with Galois connections, and transformers specified as in Figure 2.*

In order to extract the algebraic essence of DPLL(T), one can view the algorithm in terms of two synergistic strategies: (i) DPLL(T) uses γ-complete deduction to obtain a precise representation of models, and then uses $\bot$-complete reduction to check emptiness; (ii) DPLL(T) uses case splits (and learning) to resolve imprecision. It is important to see that these two strategies are independent. To illustrate, consider computing the γ-complete transformer $\mathsf{BSkelModels}$ explicitly, e.g., using BDDs instead of a case split procedure.

Theorem 2. *For an abstract* DPLL(T) *domain* $\mathsf{ADPLL}(\mathsf{BS},\mathsf{TS})$ *where* bded^* *is* $\gamma_\bot$*-complete and* tded^* *is a* $\bot$*-complete reduction, it holds that* φ *is satisfiable exactly if* $\mathsf{gfp}\ \mathsf{deduce}_\varphi \neq \bot$.

This property may be hard to achieve in practice unless an expensive abstraction is chosen for BS. In this case, case analysis with splitting (or other techniques such as clause learning) can be employed. We model these algorithms abstractly as procedures that provide decompositions of elements into precise cases. For a more detailed account, consider [7,14,8].

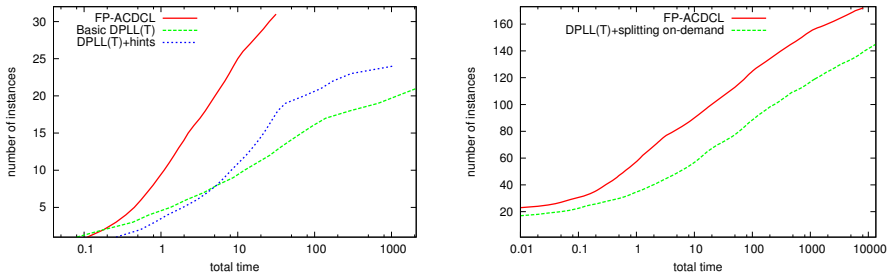
Definition 21. *A* $\gamma_\bot$-precise decomposition *is a function* $\mathsf{dc} : \mathsf{BS} \to \wp(\mathsf{BS})$ *s.t. for all elements* $\mathsf{be} \in \mathsf{BS}$ *it holds that (i)* $\mathsf{dc}(\mathsf{be})$ *is finite, (ii)* $\gamma_{\mathsf{BS}}(\mathsf{be}) \subseteq \bigsqcup\{\gamma(\mathsf{be}') \mid \mathsf{be}' \in \mathsf{dc}(\mathsf{be})\}$ *and (iii) for any* $\mathsf{bded}' \in \mathsf{dc}(\mathsf{bded})$ *the transformer* bded^* *is* $\gamma_\bot$*-complete at* bded'.

Splitting or learning-based algorithms can be viewed to generate this decomposition on demand. For an element $\mathsf{be}' \in \mathsf{BS}$, we denote by $\mathsf{deduce}_{\varphi,\mathsf{be}'}$ the function $\lambda(\mathsf{be},\mathsf{te}).\ \mathsf{deduce}_\varphi(\mathsf{be}' \sqcap \mathsf{be},\mathsf{te})$.

Theorem 3. *For an abstract* DPLL(T) *domain* $\mathsf{ADPLL}(\mathsf{BS},\mathsf{TS})$ *with* $\gamma_\bot$*-precise decomposition function* dc *and* $\bot$*-complete reduction* tded*, it holds that* φ *is satisfiable exactly if there exists a* $\mathsf{be} \in \mathsf{dc}(\top)$ *such that* $\mathsf{gfp}\ \mathsf{deduce}_{\varphi,\mathsf{be}} \neq \bot$.

Unifying Base and Theory Reasoning. An interesting consequence of the algebraic view of DPLL(T) is that we can consider architectures of the form $\mathsf{ADPLL}(\mathsf{TS},\mathsf{TS})$, which perform all steps of the algorithm directly over TS. We refer to this strategy as Abstract Conflict Driven Clause Learning (ACDCL), it is developed in detail in [8]. We present experiments in this section, based on the FP-ACDCL solver [14], an SMT solver for floating-point logic.

In DPLL(T), the vocabulary of the primary solver is limited by the structure of the formula. This can cause suboptimal performance, which is the reason why refinements of DPLL(T) introduce fresh propositions at certain points when needed. We will consider *splitting on demand* [2], which allows the introduction of new propositions during case splits, to model the effect of decision making directly in the theory.



(a) (b)

Fig. 3. Experimental results

Comparing ACDCL and DPLL(T). We present two experiments: (i) A comparison of classic DPLL(T) and ACDCL on set of hand-crafted formulae, in which the vocabulary restrictions of DPLL(T) cause enumeration behaviour. (ii) A comparison of DPLL(T) with splitting on demand and ACDCL on a set conjunctive formulae that require splitting within the theory for completeness. It is important to note that the benchmarks are specifically chosen to illustrate some limitations of DPLL(T), which can be overcome in the algebraic framework advocated in this paper. To compare against classic DPLL(T), we have integrated FP-ACDCL as a black-box theory solver in the MATHSAT5 SMT solver [13].

An example of a formula (parametrised by N) used in experiment (i) is below.

$$((x = 1) \vee \ldots \vee (x = N)) \wedge ((y = 1) \vee \ldots \vee (y = N)) \wedge ((x + y < 0) \vee (x + y > 2N))$$

Classic DPLL(T) generates lemmas only in terms of the propositions in the Boolean skeleton. In FP-ACDCL, lemmas are directly inferred over disjunctions of interval constraints, independent of whether they occur in the formula or not.

The results of the comparison are given in Figure 3 (a), which plots the number of solved instances against total execution time for FP-ACDCL and DPLL(T). To boost the power of classic DPLL(T) we experimented with a variant in which FP-ACDCL provides hints to the SAT solver: At every theory conflict, we introduce a set of propositions corresponding to the theory deductions leading up to the conflict. Although this variant is a significant improvement over default DPLL(T), it still performs much worse than FP-ACDCL.

For the second set of experiments, we have used the benchmark problems from [14]. The formulae in this set are simple conjunctions of atoms, but they require a significant amount of case splits in the interval domain. The plot in Figure 3 (b) compares FP-ACDCL and splitting-on-demand. The results show that performing case splits directly in the interval domain is more effective than splitting-on-demand. When generating lemmas during conflict analysis, FP-ACDCL can use conflict generalisation [14] to improve the strength of learnt lemmas. We attribute the faster runtime of FP-ACDCL to the better quality of the resulting learnt lemmas.

References

1. Badban, B., van de Pol, J., Tveretina, O., Zantema, H.: Generalizing DPLL and satisfiability for equalities. Inf. Comput. 205(8) (2007)
2. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on Demand in SAT Modulo Theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006)
3. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability. IOS Press (2009)
4. Cotton, S.: Natural Domain SMT: A Preliminary Assessment. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 77–91. Springer, Heidelberg (2010)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)
6. Cousot, P., Cousot, R., Mauborgne, L.: The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 456–472. Springer, Heidelberg (2011)
7. D'Silva, V., Haller, L., Kroening, D.: Satisfiability Solvers Are Static Analysers. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 317–333. Springer, Heidelberg (2012)
8. D'Silva, V., Haller, L., Kroening, D.: Abstract Conflict Driven Clause Learning. In: POPL (to apppear, 2013)
9. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric Bounds Analysis with Conflict-Driven Learning. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 48–63. Springer, Heidelberg (2012)
10. 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)
11. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. JSAT 1(3-4) (2007)
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
13. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8 (2012)
14. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding floating-point logic with systematic abstraction. In: FMCAD (2012)
15. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: POPL (2010)
16. Jin, H., Somenzi, F.: Strong conflict analysis for propositional satisfiability. In: DATE (2006)
17. Jovanović, D., de Moura, L.: Cutting to the Chase Solving Linear Integer Arithmetic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 338–353. Springer, Heidelberg (2011)
18. Jovanović, D., de Moura, L.: Solving Non-linear Arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
19. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to Richer Logics. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)

20. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
21. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). JACM 53 (2006)
22. Thakur, A., Reps, T.: A Method for Symbolic Computation of Abstract Operations. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 174–192. Springer, Heidelberg (2012)
23. Thakur, A., Reps, T.: A Generalization of Stålmarck's Method. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 334–351. Springer, Heidelberg (2012)

All for the Price of Few
(Parameterized Verification through View Abstraction)

Parosh Aziz Abdulla[1], Frédéric Haziza[1], and Lukáš Holík[1,2]

[1] Uppsala University, Sweden
[2] Brno University of Technology, Czech Republic

Abstract. We present a simple and efficient framework for automatic verification of systems with a parameteric number of communicating processes. The processes may be organized in various topologies such as words, multisets, rings, or trees. Our method needs to inspect only a small number of processes in order to show correctness of the whole system. It relies on an abstraction function that views the system from the perspective of a fixed number of processes. The abstraction is used during the verification procedure in order to dynamically detect cut-off points beyond which the search of the state space need not continue. We show that the method is complete for a large class of well quasi-ordered systems including Petri nets. Our experimentation on a variety of benchmarks demonstrate that the method is highly efficient and that it works well even for classes of systems with undecidable verification problems.

1 Introduction

We address verification of safety properties for *parameterized systems* that consist of arbitrary numbers of components (processes) organized according to a regular pattern. The task is to perform *parameterized verification*, i.e., to verify correctness regardless of the number of processes. This amounts to the verification of an infinite family; namely one for each possible size of the system. The term *parameterized* refers to the fact that the size of the system is (implicitly) a parameter of the verification problem. Parameterized systems arise naturally in the modeling of mutual exclusion algorithms, bus protocols, distributed algorithms, telecommunication protocols, and cache coherence protocols. For instance, the specification of a mutual exclusion protocol may be parameterized by the number of processes that participate in a given session of the protocol. In such a case, it is interesting to verify correctness regardless of the number of participants in a particular session. As usual, the verification of safety properties can be reduced to the problem of checking the reachability of a set of *bad configurations* (the set of configurations that violate the safety property).

Existing approaches. An important approach to parameterized verification has been *regular model checking* [25,5,9] in which regular languages are used as symbolic representations of infinite sets of system configurations, and automata-based techniques are employed to implement the verification procedure. The main problem with such techniques is that they are heavy since they usually rely on several layers of computationally expensive automata-theoretic constructions, in many cases leading to a state space

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 476–495, 2013.

explosion that severely limits their applicability. Another class of methods analyze *approximated* system behavior through the use of abstraction techniques. Such methods include *counter abstraction* [22,30], *invisible invariant* generation [6,31], *environment abstraction* [11], and *monotonic abstraction* [3] (see Section 7).

In a similar manner to [24], this work is inspired by a strong empirical evidence that parameterized systems often enjoy a *small model property*. More precisely, analyzing only a small number of processes (rather than the whole family) is sufficient to capture the reachability of bad configurations. On the one hand, bad configurations can often be characterized by minimal conditions that are possible to specify through a fixed number of *witness* processes. For instance, in a mutual exclusion protocol, a bad configuration contains *two* processes in their critical sections; and in a cache coherence protocol, a bad configuration contains *two* cache lines in their *exclusive* states. In both cases, having the two witnesses is sufficient to make the configuration bad (regardless of the actual size of the configuration). On the other hand, it is usually the case that such bad patterns (if existing) appear already in small instances of the system, as observed in our experimental section.

Our approach. We introduce a method that exploits the small model property, and performs parameterized verification by only inspecting a small set of *fixed* instances of the system. Furthermore, the instances that need to be considered are often small in size (typically three or four processes) which allows for a very efficient verification procedure. The framework can be applied uniformly to generate *fully automatic* verification algorithms for wide classes of parameterized systems including ones that operate on linear, ring, or tree-like topologies, or systems that contain unbounded collections of anonymous processes (the latter class is henceforth referred to as having a *multi-set* topology).

At the heart of the method is an operation that allows to detect *cut-off* points beyond which the verification procedure need not continue. Intuitively, reaching a cut-off point means that we need not inspect larger instances of the system: the information collected so far during the exploration of the state space allows us to conclude safely that no bad configurations will occur in the larger instances. The cut-off analysis is executed *dynamically* in the sense that it is performed on-the-fly during the verification procedure itself. It is based on an abstraction function, called *view abstraction*, parameterized by a constant k, and it approximates a configuration by the set of all its projections containing at most k processes. We call the sub-configurations *views*. For instance, when a configuration is a word of process states (represented as an array of processes), its abstraction is the set of all its subwords of length at most k. Furthermore, for a given set of views X, its concretization, denoted as $\gamma_k(X)$, is the set of configurations (of any size) for which *all* their views belong to X.

The verification method performs two search procedures in parallel. The first performs a standard (explicit-state) forward reachability analysis trying to find a bad configuration among system configurations of size k (for some natural number k). If a bad configuration is encountered then the system is not safe. The second procedure performs a *symbolic* forward reachability analysis in the abstract domain of sets of views of size at most k. When the computation terminates, it will have collected an over-approximation of all views of size up to k of all reachable configurations (of all sizes).

If there is no bad configuration in the concretization of this set, then a cut-off point has been found and the system can be claimed safe. If neither of the parallel procedures reaches a conclusion during iteration k, the value of k is increased by one (thus increasing the precision of the abstraction). Notice that the abstract search requires computing the *abstract post-image* of a set X of views of size at most k, which is the set X' of views (of size at most k) of successors of $\gamma_k(X)$. Obviously, this cannot be performed straightforwardly since the set of configurations $\gamma_k(X)$ is infinite. A crucial contribution of the paper is to show that, for all the classes of parameterized systems that we consider, it is sufficient to only compute successors of configurations from $\gamma_k(X)$ that are of the size at most $k+\ell$, where ℓ is a small constant, typically 1. Intuitively, the reason is that the precondition for firing a transition is the presence of a *bounded* number of processes in certain states. The views need only to encompass these processes in order to determine the successor view. This property is satisfied by a wide class of concurrent systems including the ones we consider in this paper. For instance, in rendez-vous communication between a pair of processes, the transition is conditioned by the states of *two* processes; in broadcast communication, *one* process initiates the transition (while the other processes may be in any state); in existential global transitions (see below), we need *two* processes, namely the witness and the process performing the transition; in Petri nets, the number of required processes is bounded by the in-degree of the transitions (which is fixed for a given Petri net), etc. We will show formally that this property is satisfied by all the types of transitions we consider.

Applications. We have instantiated the method to obtain automatic verification procedures for four classes of parameterized systems, namely systems where the processes are organized as arrays, rings, trees, or multisets. Each instantiation is straightforward and is achieved by defining the manner in which we define the views of a configuration. More precisely, these views are (naturally) defined as subwords, cyclic subwords, subtrees, resp. subsets for the above four classes. Once the views are fixed we obtain a fully automatic procedure for all parameterized systems in the class. In the systems we consider, we allow a rich set of features, in which processes may perform local transitions, rendez-vous, broadcasts, and universally or existentially guarded transitions. In a universally guarded transition, the process checks whether the states of *all* other processes inside the system satisfy a given constraint before it performs the transition. In an existentially quantified transition, the processes checks that there is *at least one* other process satisfying the condition. Furthermore, we allow dynamic behaviors such as the creation and deletion of processes during the execution of the system.

In the basic variant of our method, we assume that existential and universal global conditions of transitions are checked atomically. The same assumption is made in many landmark works on parameterized systems (e.g. [11,31,10,5,6,29,3]). However, actual implementations of global checks are usually not atomic. They are typically implemented as for-loops ranging over indices of processes. Iterations of such a loop may be interleaved with transitions of other processes, therefore modeling the loop as an atomic transition means under-approximating the behavior of the system. Verification of systems with non-atomic global checks is significantly harder. It requires to distinguish intermediate states of a for-loop performed by a process. Their number is proportional to the number of processes in the system. Moreover, any number of processes may be

performing a for-loop at the same time. As we will show, our method can be easily adapted to this setting, while retaining its simplicity and efficiency.

Implementation. We have implemented a prototype based on the method and run it on a wide class of benchmarks, including mutual exclusion protocols on arrays (e.g., Burns', Szymanski's, and Dijkstra's protocols), cache coherent protocols (e.g., MOSI and German's protocol), different protocols on tree-like architectures (e.g. percolate, arbiter, and leader election), ring protocols (token passing), and different Petri nets.

The class of systems we consider have undecidable reachability properties, and hence our method is necessarily incomplete (the verification procedure is not guaranteed to terminate in case the safety property is satisfied). However, as shown by our experimentation, the tool terminates efficiently on all the tested benchmarks.

Completeness. Although the method is not complete in general, we show that is complete for a large class of systems, namely those that induce *well quasi-ordered* transition systems [2,1] and satisfy certain additional technical requirements. This implies that our method is complete for e.g., Petri nets. Notice that, as evident from our experiments, the method can in practice handle even systems that are outside the class.

Outline. To simplify the presentation, we instantiate our framework in a step-wise manner. In Section 2, we introduce our model for parameterized systems operating on linear topologies and describe our verification method in Section 3. In Section 4, we describe how the framework can be extended to incorporate other kinds of transitions such as broadcast, rendez-vous, dynamic process deletion/creation, and non-atomic checks of global conditions; and to cover other classes of topologies such as ring, multiset, and tree-like structures. The completeness of our method for well quasi-ordered systems is shown in Section 5. We report on our experimental results in Section 6, and describe related work in Section 7. Finally, we give some conclusions and directions for future research in Section 8.

2 Parameterized Systems

We introduce a standard notion of a parameterized system operating on a linear topology, where processes may perform local transitions or universally/existentially guarded transitions (this is the standard model used e.g. in [31,11,3,29]).

A parameterized system is a pair $\mathcal{P} = (Q, \Delta)$ where Q is a finite set of *local states* of a process and Δ is a set of *transition rules* over Q. A transition rule is either *local* or *global*. A local rule is of the form $s \to s'$, where the process changes its local state from s to s' independently of the local states of the other processes. A global rule is of the form **if** $\mathbb{Q} j \circ i : S$ **then** $s \to s'$, where $\mathbb{Q} \in \{\exists, \forall\}$, $\circ \in \{<, >, \neq\}$ and $S \subseteq Q$. Here, the ith process checks also the local states of the other processes when it makes the move. For instance, the condition $\forall j < i : S$ means that "for every j such that $j < i$, the jth process should be in a local state that belongs to the set S"; the condition $\forall j \neq i : S$ means that "all processes except the ith one should be in local states that belong to the set S"; etc.

A parameterized system $\mathcal{P} = (Q, \Delta)$ induces a *transition system* (TS) $\mathcal{T} = (C, \to)$ where $C = Q^*$ is the set of its *configurations* and $\to \subseteq C \times C$ is the *transition relation*.

We use $c[i]$ to denote the state of the ith process within the configuration c. The transition relation $\rightarrow$ contains a transition $c \rightarrow c'$ with $c[i] = s$, $c'[i] = s'$, $c[j] = c'[j]$ for all $j : j \neq i$ iff either (i) Δ contains a local rule $s \rightarrow s'$, or (ii) Δ contains a global rule **if** $\mathbb{Q} j \circ i : S$ **then** $s \rightarrow s'$, and one of the following conditions is satisfied:

- $\mathbb{Q} = \forall$ and for all $j : 1 \leq j \leq |c|$ such that $j \circ i$, it holds that $c[j] \in S$.
- $\mathbb{Q} = \exists$ and there exists $j : 1 \leq j \leq |c|$ such that $j \circ i$ and $c[j] \in S$.

An instance of the *reachability problem* is defined by a parameterized system $\mathcal{P} = (Q, \Delta)$, a regular set $I \subseteq Q^+$ of *initial configurations*, and a set $Bad \subseteq Q^+$ of *bad configurations*. Let $\sqsubseteq$ be the usual *subword relation*, i.e., $u \sqsubseteq s_1 \ldots s_n$ iff $u = s_{i_1} \ldots s_{i_k}, 1 \leq i_1 \ldots i_k \leq n$ and $i_j < i_{j+1}$ for all $j : 1 \leq j < k$. We assume that Bad is the upward closure $\{c \mid \exists b \in B : b \sqsubseteq c\}$ of a given *finite* set $B \subseteq Q^+$ of *minimal bad configurations*. This is a common way of specifying bad configurations which often appears in practice, see e.g. the running example of Burn's mutual exclusion protocol below. We say that $c \in C$ is *reachable* iff there are $c_0, \ldots, c_l \in C$ such that $c_0 \in I$, $c_l = c$, and $c_i \rightarrow c_{i+1}$ for all $0 \leq i < l$. We use $\mathcal{R}$ to denote the set of all reachable configurations. We say that the system $\mathcal{P}$ is *safe* w.r.t. I and Bad if no bad configuration is reachable, i.e. $\mathcal{R} \cap Bad = \emptyset$.

We define the *post-image* of a set $X \subseteq C$ to be the set $post(X) := \{c' \mid c \rightarrow c' \wedge c \in X\}$. For $n \in \mathbb{N}$ and a set of configurations $S \subseteq C$, we use S_n to denote its subset $\{c \in S \mid |c| \leq n\}$ of configurations of size up to n.

Running example. We illustrate the notion of a parameterized systems with the example of Burns' mutual exclusion protocol [26]. The protocol ensures exclusive access to a shared resource in a system consisting of an unbounded number of processes organized in an array. The pseudocode of the process at the ith position of the array and the transition rules of the parameterized system are given in Figure 1. A state of the ith process consists of a program location and a value of the local variable $flag[i]$. Since the value of $flag[i]$ is invariant at each location, states correspond to locations.

A configuration of the induced transition system is a word over the alphabet $\{1, \ldots, 6\}$ of local process states. The task is to check that the protocol guarantees exclusive access to the shared resource (line 6) regardless of the number of processes. A configuration is considered to be bad if it contains two occurrences of state 6, i.e., the set of minimal bad configurations B is $\{6\,6\}$. Initially, all processes are in state 1, i.e. $I = 1^+$.

Burns(i)

1 $flag[i] := 0$;
2 **if** $\exists j < i : flag[i] = 1$ **then goto** 1;
3 $flag[i] := 1$;
4 **if** $\exists j < i : flag[i] = 1$ **then goto** 1;
5 **await** $\forall j > i : flag[j] \neq 1$;
6 $flag[i] := 0$; **goto** 1 CS

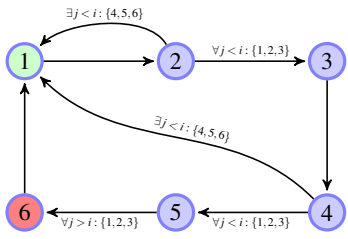


Fig. 1. Pseudocode and transition rules of Burns' protocol

3 Verification Method

In this section, we describe our verification method instantiated to the case of parameterized systems described in Section 2. First, we describe the abstraction we use, then we present the procedure.

3.1 View Abstraction

We abstract a configuration c by a set of *views* each of which is a subword of c. The *abstraction function* $\alpha_k : C \to 2^{C_k}$ maps a configuration c into the set $\alpha_k(c) = \{v \in C_k \mid v \sqsubseteq c\}$ of all its views (subwords) of size up to k. We lift α_k to sets of configurations as usual. For every $k \in \mathbb{N}$, the *concretization* function $\gamma_k : 2^{C_k} \to 2^C$ inputs a set of views $V \subseteq C_k$, and returns the set of configurations that can be reconstructed from the views in V. In other words, $\gamma_k(V) = \{c \in C \mid \alpha_k(c) \subseteq V\}$.

Abstract post-image. As usual, the *abstract post-image* of a set of views $V \subseteq C_k$ is defined as $Apost_k(V) = \alpha_k(post(\gamma_k(V)))$. Computing $Apost_k(V)$ is a central component of our verification procedure. It cannot be computed straightforwardly since the set $\gamma_k(V)$ is typically infinite. As a main contribution of the paper, we show that it is sufficient to consider only those configurations in $\gamma_k(V)$ whose sizes are up to $k+1$. There are finitely many such configurations, and hence their post-image can be computed. Formally, for $\ell \geq 0$, we define $\gamma_k^\ell(V) := \gamma_k(V) \cap C_\ell$ and show the following *small model* lemma for the class of systems of Section 2. We will show similar lemmas for the other classes of systems that we present in the later sections.

Lemma 1. *For any $k \in \mathbb{N}$ and $X \subseteq C_k$, $\alpha_k(post(\gamma_k(X))) \cup X = \alpha_k(post(\gamma_k^{k+1}(X))) \cup X$.*

The property of the transition relation which allows us to prove the lemma is that, loosely speaking, the transitions have *small preconditions*. That is, there is a transition that can be fired from a configuration c and generate a view $v \in C_k$ only if c contains a certain view v' of some limited size, here up to $k+1$.

Running Example. Consider for instance the set $V = \{1,2,3,4,6,12,16,32,34,42\} \subseteq C_2$ of views of Burns' protocol. We will illustrate that we need to reason only about configurations of $\gamma_2(V)$, which are of size at most 3, to decide which views belong to $Apost_2(V)$.

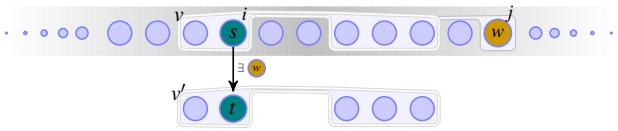
Take the existentially guarded transition $2 \to 1$. It can be fired only from configurations that contain 2 together with a witness from $\{4,5,6\}$ on the left. $Apost_2(V)$ contains the view 31 since $\gamma_2^{2+1}(V)$ contains 342 from where the existential transition $2 \to 1$ can be fired. (342 belongs to $\gamma_2(V)$ because all its views 2, 3, 4, 32, 34, and 42 are in V). It does not contain the view 22 since 12 cannot be completed by the needed witness (12 cannot be extended by, e.g., 6 since V does not contain 26 and 62).

Consider now the universally guarded transition $2 \to 3$. The transition can be fired only from configurations that contain 2. Since $2 \to 3$ can be fired on $32 \in \gamma_2(V)$, $Apost_2(V)$ contains 33. But it does not contain the view 43 since the universal guard prevents firing $2 \to 3$ on configurations containing 42.

Proof. We present the part of the proof of Lemma 1 which deals with existentially guarded transitions. The parts dealing with local and universally guarded transitions are simpler and are moved to the appendix. We will show that for any configuration $c \in \gamma_k(V)$ of size $m > k+1$ such that there is a transition $c \to c'$ induced by an existentially guarded rule $r \in \Delta$ with $v' \in \alpha_k(c')$, the following holds: Either $v' \in V$ or there is a configuration $d \in \gamma_k(V)$ of size at most $k+1$ with a transition $d \to d'$ induced by r with $v' \in \alpha_k(d')$.

A subset of positions $p = \{i_1, \ldots, i_l\} \subseteq \{1, \ldots, n\}, l \leq k$, with $i_1 < \ldots < i_l$ of a configuration $c = s_1 \ldots s_n$ defines the view $view(c,p) = s_{i_1} \ldots s_{i_l}$ of c. By definition, v' equals $view(c',p)$ for some $p \subseteq \{1, \ldots, m\}$. Let v be $view(c,p)$. Notice that since $c \in \gamma_k(V)$, any view of c of size at least k belongs to $\gamma_k(V)$. Therefore also $v \in \gamma_k(V)$. Let $1 \leq i \leq m$ be the index of the position in which c' differs from c. If $i \notin p$, then $v = view(c,p) = view(c',p) = v'$. In this case, we trivially have $v' \in V$. We can take $d = v$ and $d' = v'$.

Assume now that $i \in p$. Let r be the rule **if** $\exists j \circ i : S$ **then** $s \to t$ where $\circ \in \{<, >, \neq\}$. There are two cases: 1) there is a witness w from S at some position $j \in p$ enabling the transition $c \to c'$. Then v still contains the witness on an appropriate position needed to fire r. Therefore $v \to v'$ is a transition of the system induced by r, and we can take $d = v$ and $d' = v'$. 2) no witness enabling the transition $c \to c'$ is at a position $j \in p$. Then there is no guarantee that $v \to v'$ is a transition of the system. However, the witness enabling the transition $c \to c'$ is at some position $j \in \{1, \ldots, m\}$. We will create a configuration of size at most $k+1$ by including this position j to v, as illustrated in the figure. Let $p' = p \cup \{j\}$. Then $view(c,p') \to view(c',p')$ is a transition of the system induced by r since $view(c,p')$ contains both s and a witness from S at an appropriate position. We clearly have that $v' \in \alpha_k(view(c',p'))$. We also have that $view(c,p') \in \gamma_k(V)$ since $view(c,p') \sqsubseteq c$ and $c \in \gamma_k(V)$. We may therefore take $d = view(c,p')$ and $d' = view(c',p')$. □



3.2 Procedure

Our verification procedure for solving an instance of the verification problem defined in Section 2 is described in Algorithm 1. It performs two search procedures in parallel. Specifically, it searches for a bad configuration reachable from initial configurations of size k; and it searches for a cut-off point k where it derives a set of views $V \subseteq C_k$ such that

(i) V is an invariant for the instances of the system (that is, $\mathcal{R} \subseteq \gamma_k(V)$ and $Apost_k(V) \subseteq V$), and
(ii) which is sufficient to prove $\mathcal{R}$ safe (that is, $\gamma_k(V) \cap Bad = \emptyset$).

Algorithm 1. Verification Procedure

```
1 for k := 1 to ∞ do
2     if R_k ∩ Bad ≠ ∅ then return Unsafe
3     V := μX.α_k(I) ∪ Apost_k(X)
4     if γ_k(V) ∩ Bad = ∅ then return Safe
```

For a given k, an invariant V is computed on line 3. Notice that V is well-defined since $\gamma_k, post, \alpha_k$ and hence also $Apost_k$ are monotonic functions for all $k \in \mathbb{N}$ (w.r.t. $\subseteq$). Lemma 2 guarantees that V is indeed an invariant:

Lemma 2. *For any $i \in \mathbb{N}$ and $X \subseteq C_i$, $\alpha_i(I) \subseteq X \wedge Apost_i(X) \subseteq X \implies \alpha_i(\mathcal{R}) \subseteq X$.*

If the system is unsafe, the search on line 2 will eventually discovers a bad configuration. The cut-off condition is tested on line 4. If the test does not pass, then we do not know whether the system is indeed unsafe or whether the analysis has hit a spurious counterexample (due to a too liberal abstraction). Therefore, the algorithm increases precision of the abstraction by increasing k and reiterating the loop. An effective implementation of the procedure requires carrying out the following steps:

1. *Computing the abstraction $\alpha_k(I)$ of initial configurations.* This step is usually easy. For instance, in the case of Burns' protocol, all processes are initially in state 1, hence $\alpha_k(I)$ contains only the words $1^l, l \leq k$. Generally, I is a (very simple) regular set, and $\alpha_k(I)$ is computed using a straightforward automata construction.
2. *Computing the abstract post-image.* Thanks to Lemma 1, the abstract post-image can be computed by applying γ_k^{k+1} (which yields a finite set), *post*, and α_k (in that order).
3. *Evaluating the test $\gamma_k(V) \cap Bad = \emptyset$.* Since *Bad* is the upward closure of a finite set B, the test can be carried out by testing whether there is $b \in B$ such that $\alpha_k(b) \subseteq V$.
4. *Exact reachability analysis.* Line 2 requires the computation of $\mathcal{R}_k$. Since $\mathcal{R}_k$ is finite, this can be done using any procedure for exact state space exploration.

Since the problem is generally undecidable, existence of k for which the test on line 4 succeeds for a safe system cannot be guaranteed and the algorithm may not terminate. However, as discussed in Section 5, such a guarantee can be given under the additional requirement of monotonicity of transition relation w.r.t. a well-quasi ordering. The method terminates otherwise for all our examples discussed in Section 6, many of which are *not* well quasi-ordered.

Running example. When run on Burns' protocol, Algorithm 1 starts by computing $\mathcal{R}_1 = \{1,\ldots,6\}$. Because $\mathcal{R}_1$ does not contain any bad configurations, the algorithm moves onto computing the fixpoint V_1 of line 3. The iteration starts with $X = \alpha_1(I) = \{1\}$ and continues until $X = V_1 = \{1,\ldots,6\}$. The test on line 4 subsequently fails since $\gamma_1(V_1)$ contains 66. Since both tests fail, the first iteration does not allow us to conclude whether the protocol is safe or not, so the algorithm increases the precision of the abstraction by increasing k.

In the second iteration with $k = 2$, $\mathcal{R}_2$ is still safe. The fixpoint computation starts with $X = \alpha_2(I) = \{1, 11\}$. When $Apost_2$ is applied on $\{1, 11\}$, we first construct the set $\gamma_2^{2+1}(\{1, 11\})$ which contains the extension 111 of 11, 11 and 1. Their successors are $2, 12, 21$, and $112, 121, 211$, which are abstracted into $\{1, 2, 11, 12, 21\}$. The fixpoint computation continues with $X = \{1, 2, 11, 12, 21\}$ and constructs the concretization $\gamma_2^3(X) = X \cup \{112, 121, 211\}$. Their successors are $2, 3, 12, 21, 22, 31, 13$, and 122, 212, 221, 113, 131, 311 which are abstracted into the views $1, 2, 3, 11, 12, 21, 22, 31, 13$. The next iteration will start with $X = \{1, 2, 3, 11, 12, 21, 22, 13, 31\}$. The computation reaches, after 8 further iterations, the fixpoint $X = V_2$ which contains all words from $\{1, \ldots, 6\} \cup \{1, \ldots, 6\}^2$ except 65 and 66. This set satisfies the assumptions of Lemma 2, and hence it is guaranteed to contain all views (of size at most 2) of all reachable configurations of the system. Since the view 66 is not present (recall $\alpha_2(Bad) = \{6, 66\}$), no reachable configuration of the system is bad. The algorithm reached the cut-off point $k = 2$ of Burns' protocol, and the system is proved safe.

4 Extensions

In this section, we describe how to extend the class of parameterized systems that we presented in Section 2. The extensions are obtained 1) by extending the types of transition rules that we allow, 2) by replacing transitions with atomically checked global conditions by more realistic for-loops, and 3) by considering topologies other than the linear ones. As we shall see below, the extensions can be handled by our method with straightforward extensions of the method of Section 3.

4.1 More Communication Mechanisms

Broadcast. In a broadcast transition, an arbitrary number of processes change states simultaneously. A broadcast rule is a pair $(s \to s', \{r_1 \to r_1', \ldots, r_m \to r_m'\})$. It is deterministic in the sense that $r_i \neq r_j$ for $i \neq j$. The broadcast is initiated by a process, called the *initiator*, which triggers the transition rule $s \to s'$. Together with the initiator, an arbitrary number of processes, called the *receptors*, change state simultaneously. More precisely, if the local state of a process is r_i, then the process changes its local state to r_i'. Processes whose local states are different from $s, r_1, \ldots r_m$ remain passive during the broadcast. Formally, the broadcast rule induces transitions $c \to c'$ of $\mathcal{T}$ where for some $i : 1 \leq i \leq |c|$, $c[i] = s$, $c'[i] = s'$, and for each $j : 1 \leq j \neq i \leq |c|$, if $c[j] = r_k$ (for some k) then $c'[j] = r_k'$, otherwise $c[j] = c'[j]$.

In a similar manner to globally guarded transitions, broadcast transitions have small preconditions. Namely, to fire a transition, it is enough that an initiator is present in the transition. More precisely, for parameterized systems with local, global, and broadcast transitions, Lemma 1 still holds (in the proof of Lemma 1, the initiator is treated analogously to a witness of an existential transition). Therefore, the verification method from Section 3 can be used without any change.

Rendez-vous. In rendez-vous, multiple processes change their states simultaneously. A *simple rendez-vous* transition rule is a tuple of local rules $\delta = (r_1 \to r_1', \ldots, r_m \to$

$r'_m), m > 1$. Multiple occurrences of local rules with the same source state r in the tuple does not mean non-determinism, but that the rendez-vous requires multiple occurrences of r in the configuration to be triggered. Formally, the rule induces transitions $c \to c'$ of $\mathcal{T}$ such that there are $i_1, \ldots, i_m$ with $i_j \neq i_k$ for all $j \neq k$, such that $c[i_1] \cdots c[i_m] = r_1 \cdots r_m$, $c'[i_1] \cdots c'[i_m] = r'_1 \cdots r'_m$, and $c'[\ell] = c[\ell]$ if $\ell \notin \{i_1, \ldots, i_m\}$.

Additionally, we define a *generalized rendez-vous (or just rendez-vous) transition rules* in order to model *creation and deletion* of processes and also Petri net transitions that change the number of tokens in the net. A generalized rendez-vous rule δ is as a simple rendez-vous rule, but it can in addition to the local rules contain two types of special rules: of the form $\bullet \to r, \bullet \notin Q$ (acting as a placeholder), which are used to model creation of processes, and of the form $r \to \bullet$, which are used to model deletion of processes. When a generalized rendez-vous rule is fired, the starting configuration is first enriched with $\bullet$ symbols in order to facilitate creation of processes by the rule $\bullet \to r$, then the rule is applied as if it was a simple rendez-vous rule, treating $\bullet$ as a normal state (states of the processes that are to be deleted are rewritten to $\bullet$ by the rules $r \to \bullet$). Finally, all occurrences of $\bullet$ are removed. Formally, a generalized rendez-vous rule induces a transition $c \to c'$ if there is $c_\bullet \in \{\bullet\}^* c[1]\{\bullet\}^* \cdots \{\bullet\}^* c[|c|]\{\bullet\}^*$ such that $c_\bullet \to c'_\bullet$ is a transition of the system with states $Q \cup \{\bullet\}$ induced by δ (treated as a simple rendez-vous rule), and c' arises from $c'_\bullet$ by erasing all occurrences of $\bullet$.

Rendez-vous transitions have small preconditions, but unlike existentially quantified transitions, firing a transition may require presence of more than two (but still a fixed number) processes in certain states (the number is the arity of the transition). It essentially corresponds to requiring the presence of more than one witness. This is why Lemma 1 holds here only in the weaker variant:

Lemma 3. *Let Δ contain rules of any previously described type (i.e., local, global, broadcast, rendez-vous), and let $m+1$ is the largest arity of a rendez-vous rule in Δ. Then, for any k and $V \subseteq C_k$, $\alpha_k(post(\gamma_k(V))) \cup V = \alpha_k(post(\gamma_k^{k+m}(V))) \cup V$.*

Global variables. Communication via shared variables is modeled using a special process, called *controller*. Its local state records the state of all shared variables in the system. A configuration of a system with global variables is then a word $s_1 \ldots s_n c$ where $s_1, \ldots, s_n$ are the states of individual processes and c is the state of the controller. An individual process can read and update a shared variable. A read is modeled by a rendez-vous rule of the form $(s \to s', c \to c)$ where c is a state of the controller and s, s' are states of the process. An update is modeled using a rendez-vous rule $(s \to s', c \to c')$.

To verify systems with shared variables of finite domains, we use a variant of the abstraction function which always keeps the state of the controller in the view. Formally, for a configuration wc where $w \subseteq Q^+$ and c is the state of the controller, α_k returns the set of words vc where v is a subword of w of length at most k. The concretization and abstract-post image are then defined analogously as before, based on α_k, Lemma 1 and Lemma 2 still hold. The method of Section 3 can be thus used in the same way as before.

Another type of global variable is a *process pointer*, i.e., a variable ranging over process indices. This is used, e.g., in Dijkstra's mutual exclusion protocol. A process pointer is modeled by a local Boolean flag p for each process state. The value of p is

true iff the pointer points to the process (it is true for precisely one process in every configuration). An update of the pointer is modeled by a rendez-vous transition rule which sets to *false* the flag of the process currently pointed to by the pointer and sets to *true* the flag of the process which is to become the target of the pointer.

4.2 Transitions That Do not Preserve Size

We now discuss the case when the transition relation does not preserve size of configurations, which happens in the case of generalised rendez-vous. $\mathcal{R}_k$ then cannot be computed straightforwardly since computations reaching configurations of the size up to k may traverse configurations of larger sizes. Therefore, similarly as in [21], we only consider runs of the system visiting configurations of the size up to k. That is, on line 2 of Algorithm 1, instead of computing $\mathcal{R}_k = \mu X.\, I_k \cup post(X)$, we compute its under-approximation $\mu X.\, (I \cup post(X)) \cap C_k$. The computation terminates provided that C_k is finite. The algorithm is still guaranteed to return Unsafe if a configuration in *Bad* is reachable, since then there is $k \in \mathbb{N}$ such that the bad configuration is reachable by a finite path traversing configurations of the size at most k.

4.3 Non-atomic Global Conditions

We extend our method to handle systems where global conditions are not checked atomically. We replace both existentially and universally guarded transition rules by a simple variant of a for-loop rule:

$$\textbf{if foreach } j \circ i : S \textbf{ then } q \to r \textbf{ else } q \to s$$

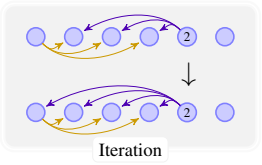
where $q, r, s \in Q$ is resp. a source state, a target state, and an escape state, $\circ \in \{<, >, \neq\}$, and $S \subseteq Q$ is a *condition*. For instance, line 2 of Burns' protocol would be replaced by **if foreach** $j < i : \{1,2,3\}$ **then** $2 \to 3$ **else** $2 \to 1$.

The semantics of a system with for-loop rules is defined as an extension of the transition system from Section 2. Configurations are extended with a binary relation over their positions, that is, a configuration is now a pair $(c, \checkmark)$ where c is a word over Q and $\checkmark$ is a binary relation over its positions $\{1, \ldots, |c|\}$. The relation $\checkmark$ is used to encode intermediate states of for-loops. Intuitively, a process at position i performing a for-loop puts (i, j) into $\checkmark$ to mark that it has processed the position j.

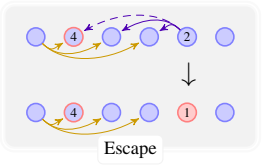
Formally, a parameterized system $\mathcal{P} = (Q, \Delta)$ which includes for-loop rules induces a transition system $\mathcal{T} = (C, \to)$ where $C \subseteq Q^+ \times (\mathbb{N} \times \mathbb{N})$. For technical convenience, we assume that a source of a for-loop rule in Δ is not a source of any other rule in Δ.[1] Then every for-loop rule **if foreach** $j \circ i : S$ **then** $q \to r$ **else** $q \to s$ induces transitions $t = (w, \checkmark) \to (w', \checkmark')$ with $w[i] = q$ for some $i : 1 \leq i \leq |w|$ which may be of the following three forms: (illustrated using the aforementioned example rule from Burn's protocol).

[1] Without this restriction, the state of a process would have to contain additional information recording which for-loop is the process currently performing. Note that the restriction does not limit the modeling power of the formalism. Any potential branching may be moved to predecessors of the sources of the for-loop.

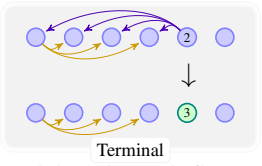
Iteration: The ith process checks that the state of a next unchecked process in the range is in S and marks it. That is, there is $j : 1 \leq j \leq |w|$ with $j \circ i$, $(i,j) \notin \checkmark$, $w[j] \in S$, and the resulting configuration has $w' = w$ and $\checkmark' = \checkmark \cup \{(i,j)\}$.



Escape: If the state of some process in the range which is still to be checked violates the loop condition, then the ith process may escape to the state s. That is, there is $j : 1 \leq j \leq |w|$ with $j \circ i$, $(i,j) \notin \checkmark$, and $w[j] \notin S$. The resulting configuration has $w'[k] = w[k]$ for all $k \neq i$ and $w[i] = s$. The execution of the for-loop ends and the marks of process i are reset, i.e., $\checkmark' = \checkmark \setminus \{(i,k) \mid k \in \mathbb{N}\}$.



Terminal: When the states of all processes from the range have been successfully checked, the for-loop ends and the ith process moves to the terminal state r. That is, if there is no $j : 1 \leq j \leq |w|$ with $j \circ i$ and $(i,j) \notin \checkmark$, then $w'[k] = w[k]$ for all $k \neq i$, $w'[i] = r$, and $\checkmark' = \checkmark \setminus \{(i,k) \mid k \in \mathbb{N}\}$.



Other rules behave as before on the w part of configurations and they do not influence the $\checkmark$ part. That is, a local, broadcast, or rendez-vous rule induces transitions $(w,\checkmark) \to (w',\checkmark)$ where $w \to w'$ is a transition induced by the rule as described in Section 2.

Verification. To verify systems with for-loop rules using our method, we define an abstraction α_k. Intuitively, we view a configuration $c = (w,\checkmark)$ as a graph with vertices being the positions of w and edges being defined by (i) the ordering of the positions and (ii) the relation $\checkmark$. The vertices are labeled by the states of processes at the positions. $\alpha_k(c)$ then returns the set of subgraphs of c where every subgraph contains a subset of at most k vertices of c (positions of w) and the maximal subset of edges of c adjacent with the chosen vertices.

Formally, given a configuration $c = (w,\checkmark)$, $\alpha_k(c)$ is the set of views $v = (w',\checkmark') \in C$ of size at most k (i.e., $|w'| = l \leq k$) such that there exists an injection $\rho : \{1,\ldots l\} \to \{1,\ldots,|w|\}, l \leq k$ where for all $i,j : 1 \leq i,j \leq l$:

- $i < j$ iff $\rho(i) < \rho(j)$,
- $w'[i] = w[\rho(i)]$ (i.e., $w' \sqsubseteq w$), and
- $(i,j) \in \checkmark'$ iff $(\rho(i),\rho(j)) \in \checkmark$.

The notions of concretization and abstract post-image are defined in the same manner as in Section 3 based on based on α. Lemma 1 holds here in the same wording (as shown in the appendix). Thus the verification method for systems with for-loops is analogous to the method of Section 3.

4.4 Tree Topology

We extend our method to systems where configurations are trees. For simplicity, we restrict ourselves to complete binary trees.

Trees. Let N be a prefix closed set of words over the alphabet $\{0,1\}$ called *nodes* and let Q be a finite set. A (binary) *tree* over Q is a mapping $t : N \to Q$. The node ε is called the *root*, nodes that are not prefixes of other nodes are called *leaves*. For a node $v = v'i, i \in \{0,1\}$, v' is the *parent* of v, the node $v0$ is the *left child* of v and $v1$ is its *right child*. Every node $v' = vw, w \in \{0,1\}^+$ is a *descendant* of v. The *depth* of the tree is the length of the longest leaf. A tree is *complete* if all its leaves have the same length and every non-leaf node has both children. A tree $t' : N' \to Q$ is a *subtree* of t, denoted $t' \preceq t$, iff there exists a injective map $e : N' \to N$ which respects the descendant relation and labeling. That is, $t'(v) = t(e(v))$ and v is a descendant of v' iff $e(v)$ is a descendant of $e(v')$.

Parameterized systems with tree topology. The definitions for parameterized systems with a tree topology are analogous to the definitions for systems with a linear topology (Section 2). A parameterized system $\mathcal{P} = (Q, \Delta)$ induces a transition system $\mathcal{T} = (C, \to)$ where C is the set of complete trees over Q. The set Δ of transition rules is a set of *local* and *tree* transition rules. The transitions of $\to$ are obtained from rules of Δ as follows. A *local* rule is of the form $s \to s'$ and it locally changes the label of a node from s to s'. A *tree* rule is a triple $s(s_0, s_1) \to s'(s'_0, s'_1)$. The rule can be applied to a node v and its left and right children v_0, v_1 with labels s, s_0, and s_1, respectively, and it changes their labels to s', s'_0, and s'_1, respectively.

The reachability problem is defined in a similar manner to the case of linear systems. The set B of minimal bad configurations is a finite set of trees over Q, I is a regular tree-language, and *Bad* is the upward closure of B w.r.t. the subtree relation $\preceq$. In the notation C_n and $\mathcal{R}_n$, n refers to the depth of trees rather than to the length of words.

Verification. The verification method of Section 3 is easily extended to the tree topology. The text of Section 3 can be taken almost verbatim with the difference that instead of words, we manipulate complete trees, subword relation is replaced by subtree relation, and k now refers to the depth of trees rather than the length of words. That is, a view of size k is a tree of depth k and the abstraction $\alpha_k(t)$ returns all complete subtrees of depth at most k of the tree t. Concretization and abstract post-image are defined analogously as in Section 3, based on α_k. The set I may be given in the form of a tree automaton. The computation of $\alpha_k(I)$ may be then done over the structure of the tree automaton. We can compute the abstract post-image since Lemma 1 holds here in the same wording as in Section 3. The test $\gamma_k(V) \cap Bad = \emptyset$ is carried out in the same way as in Section 3 since *Bad* is an upward closure of a set B w.r.t. $\preceq$. The points 1-4 of Section 3 are thus satisfied and Algorithm 1 can be used as a verification procedure for systems with tree topology.

4.5 Ring Topology

The method can be extended also to systems with a ring topology. In a parameterized system with ring topology, processes are organized in a circular array and they synchronize by near-neighbor communication. We model system with a ring topology as systems with linear topology of Section 2, where a configuration $c \in Q^+$ is interpreted as a circular word. The set Δ may contain local and *near-neighbor* transition rules.

A near-neighbor rule is a pair $(s_1 \to s_1', s_2 \to s_2')$. It induces the transition $c \to c'$ of $\to$ if either $c = c_L\, s_1 s_2\, c_R$ and $c' = c_L\, s_1' s_2'\, c_R$ (i.e. the 2 processes are adjacent in the configuration c) or $c = s_2\, \bar{c}\, s_1$ and $c' = s_2'\, \bar{c}\, s_1'$ (i.e. the 2 processes are positioned at the end of the configuration c). The latter case covers the communication between the extremities since configurations encode circular words.

Verification. A word u is a *circular subword* of a word v, denoted $u \trianglelefteq v$, iff there are v_1, v_2 such that $v = v_1 v_2$ and $u \sqsubseteq v_2 v_1$. The only difference compared to the method for the systems with a linear topology is that the standard subword relation is in all definitions replaced by the circular subword relation $\trianglelefteq$. An equivalent of Lemma 1 holds here in unchanged wording, points 1-4 are satisfied, and Algorithm 1 is thus a verification procedure for systems with ring topology.

4.6 Multiset Topology

Systems which we refer to as systems with multiset topology are a special case of the systems with a linear topology of Section 2. Typical representatives of these systems are Petri nets, which correspond precisely to systems of Section 4 with only (generalized) rendez-vous transitions. Systems with multiset topology may contain all types of transitions including local, global, broadcast, and rendez-vous, with the exception of global transitions with the scope of indices $j > i$ and $j < i$ (i.e., only $j \neq i$ is permitted). Since the processes have no way of distinguishing their respective positions within a configuration, the notion of ordering of positions within a configuration is not meaningful and configurations can be represented as multisets.

5 Completeness for Well Quasi-Ordered Systems

In this section, will show that the scheme desribed by Algorithm 1 is complete for a wide class of well-quasi ordered systems. To state the result in general terms, we will first give some definitions from the theory of well quasi-ordered systems (c.f. [1]).

A *well quasi-ordering* (WQO) is a preorder $\preceq$ over a set S such that for every infinite sequence $s_1, s_2, \ldots$ of elements of S, there exists i and j such that $i < j$ and $s_i \preceq s_j$. The *upward-closure* $\uparrow T$ of a set $T \subseteq S$ w.r.t. $\preceq$ is the set $\{s \in S \mid \exists t \in T : t \preceq s\}$ and its *downward-closure* is the set $\downarrow T = \{s \in S \mid \exists t \in T : s \preceq t\}$. A set is *upward-closed* if it equals its upward-closure and it is *downward-closed* if it equals its downward-closure. If T is upward closed, its complement $S \setminus T$ is downward closed and, conversely, if T is downward closed, its complement is upward closed. For every upward closed set T, there exists a minimal (w.r.t $\subseteq$) set Gen such that $\uparrow Gen = T$, called *generator* of T, which is finite. If moreover $\preceq$ is a partial order, then Gen is unique.

A relation $R \subseteq S \times S$ is *monotonic* w.r.t. $\preceq$ if whenever $(s_1, s_2) \in R$ and $s_1 \preceq s_1'$, then there is s_2' with $(s_1', s_2') \in R$ and $s_2 \preceq s_2'$. Given a relation $f \subseteq S \times S$ monotonic w.r.t. $\preceq$ and a set $T \subseteq S$, it holds that if $f(T) \subseteq T$, then $f(\downarrow T) \subseteq \downarrow T$, where $f(T)$ is the image of T defined as $\{t' \mid \exists t \in T : (t, t') \in f\}$.

The reasoning in Section 3 is based on the natural notion of a size of a configuration. Its generalization is the notion of a *discrete measure* over a set S, a function $|.| : S \to \mathbb{N}$

which fulfills the property that for every $k \in \mathbb{N}$, $\{s \in S \mid |s| = k\}$ is finite. A discrete measure is necessary to obtain the completeness result as it allows enumerating elements of S of the same size. In particular, this property guarantees termination of the fixpoint computation on Line 3 of Algorithm 1. We note that the existence of a discrete measure is implied by a stronger restriction of [8] to the so called discrete transition systems.

We say that a transition system $\mathcal{T} = (C, \rightarrow)$ is *well-quasi ordered* by a WQO $\preceq \subseteq C \times C$ if $\rightarrow$ is monotonic w.r.t. $\preceq$. Given a well-quasi ordered transition system and a measure $|.| : C \rightarrow \mathbb{N}$, we define an abstraction function $\alpha_k, k \in \mathbb{N}$ such that $\alpha_k(c) = \{c' \in C \mid c' \preceq c\}$. The corresponding concretization γ_k and abstract post-image $Apost_k$ are then defined based on α_k and $|.|$ as in Section 3.1.

Lemma 2 holds here in the same wording as in Section 3. The main component of the completeness result is the following theorem.

Theorem 1. *Let $\mathcal{T} = (C, \rightarrow)$ be a well-quasi ordered transition system with a measure $|.|$. Let I be any subset of C and let Bad be upward-closed w.r.t. $\preceq$. Then, if $\mathcal{T}$ is safe w.r.t. I and Bad, then there is $k \in \mathbb{N}$ such that for $V = \mu X.\alpha_k(I) \cup Apost_k(X)$, $Bad \cap \gamma_k(V) = \emptyset$.*

Proof. Recall first that $\gamma_k, post, Apost_k, \alpha_k$ are monotonic functions w.r.t. $\subseteq$ for all $k \in \mathbb{N}$. Let *Gen* be the minimal generator of the upward closed set $C \setminus \downarrow\mathcal{R}$. We will prove that k can be chosen as $k = \max\{|c| \mid c \in Gen\}$. Such k exists because *Gen* is finite.

We first show an auxiliary claim that $\gamma_k(\alpha_k(\downarrow\mathcal{R})) \subseteq \downarrow\mathcal{R}$. Let $s \in \gamma_k(\alpha_k(\downarrow\mathcal{R}))$. For the sake of contradiction, suppose that $s \notin \downarrow\mathcal{R}$. We have that $s \in C \setminus \downarrow\mathcal{R} = \uparrow Gen$ and there is a generator $t \in Gen$ with $t \preceq s$. By the definition of k, $|t| \leq k$. Since $t \in Gen$, $t \notin \downarrow\mathcal{R}$ and hence $t \notin \alpha_k(\downarrow\mathcal{R})$. But due to this and since $t \preceq s$, we have that $s \notin \gamma_k(\alpha_k(\downarrow\mathcal{R}))$ (by the definition of γ_k) which contradicts the initial assumption and the claim is proven.

Next, we argue that $\alpha_k(\downarrow\mathcal{R})$ is stable under abstract post, that is, $Apost_k(\alpha_k(\downarrow\mathcal{R})) \subseteq \alpha_k(\downarrow\mathcal{R})$. Since $\mathcal{R}$ is stable under *post* and *post* is monotonic w.r.t. $\preceq$, we know that $\downarrow\mathcal{R}$ is stable under *post* (that is, $post(\downarrow\mathcal{R}) \subseteq \downarrow\mathcal{R}$). Then, by the definition of $Apost_k$, and by monotonicity of α_k w.r.t. $\subseteq$, we have $Apost_k(\alpha_k(\downarrow\mathcal{R})) = \alpha_k(post(\gamma_k(\alpha_k(\downarrow\mathcal{R})))) \subseteq \alpha_k(post(\downarrow\mathcal{R})) \subseteq \alpha_k(\downarrow\mathcal{R})$.

Since $\downarrow\mathcal{R}$ contains I, $\alpha_k(I) \subseteq \alpha_k(\downarrow\mathcal{R})$. $\alpha_k(\downarrow\mathcal{R})$ is thus a fixpoint of $\lambda X.\alpha_k(I) \cup Apost_k(X)$. Because V is the least fixpoint of $\lambda X.\alpha_k(I) \cup Apost_k(X)$, $V \subseteq \alpha_k(\downarrow\mathcal{R})$. From, $\mathcal{R} \cap Bad = \emptyset$ and since *Bad* is upward closed, we know that $\downarrow\mathcal{R} \cap Bad = \emptyset$. Because $\gamma_k(V) \subseteq \gamma_k(\alpha_k(\downarrow\mathcal{R})) \subseteq \downarrow\mathcal{R}$ and $\downarrow\mathcal{R} \cap Bad = \emptyset$, $\gamma_k(V) \cap Bad = \emptyset$. □

Theorem 1 guarantees that for a safe well quasi-ordered system, there exists k for which the test on line 4 of Algorithm 1 succeeds. Conversely, Lemma 2, which, as mentioned above, still holds for the general class of well-quasi ordered systems, then assures than if the test on line 2 succeeds, the system is indeed safe.

Complete algorithm. The schema described by Algorithm 1 (or its variant from Section 4.2 if the transition relation is not size-preserving) gives a complete verification procedure for a well quasi-ordered system provided that all the four steps of its for-loop can be effectively evaluated. This is guaranteed by the following requirements:

i. $\alpha_k(I)$ can be computed,
ii. the measure $|.|$ is discrete,

iii. for a configuration c, $post(c)$ and $\alpha_k(c)$ can be computed,
iv. for a finite set of views V, $\gamma_k^{k+1}(V)$ can be computed, and
v. a variant of Lemma 1 holds.

Point (i) is point 1 of Section 3. Points (iii)-(v) guarantee that we can compute abstract post-image (point 2 of Section 3). We can test $\gamma_k(V) \cap Bad = \emptyset$ (point 3 of Section 3) since due to (iii), V is always finite. Exact reachability analysis of configurations of a bounded size (point 4 of Section 3) can be carried out since we can iterate *post* due to (iii) and the iteration terminates after a finite number of steps due to (ii). Point (ii) also assures termination of the computation of the fixpoint on line 3 (V is always finite).

Overall, Algorithm 1 is a complete verification procedure for parameterized systems of Section 2 with local and existential transitions rules, broadcast and rendez-vous. The induced transition relation is indeed monotonic w.r.t. the preorder $\sqsubseteq$ which is a WQO and the length of a configuration is a discrete measure. An important subclass of such systems are Petri nets, which, as mentioned in Section 4, correspond to systems with multiset topology and generalized rendez-vous transition rules. Systems of Section 2 with universally guarded transition rules do not satisfy the assumptions: the induced transition relation is not monotonic.

6 Experimental Results

Based on our method, we have implemented a prototype in OCaml to check safety properties for a number of parameterized systems with different topologies. The examples cover cache coherence protocols, communication protocols through trees and rings and mutual exclusion protocols.

Table 1. Experimental Results

	Protocol	**Time**	k	$\lvert V\rvert$	$\lvert\gamma_k^{k+\ell}(V)\rvert$
Array	Demo (toy example)	0.01s	2	17	53
	Burns	0.01s	2	34	186
	Dijkstra	0.07s	2	93	695
	Szymanski	0.02s	2	48	264
Multiset	MOSI Coherency	0.01s	1	10	23
	German's Coherency	15.3s	6	1890	15567
Petri Net	German (simplified)	0.03s	2	43	96
	BH250	2.85s	2	503	503
	MOESI Coherency	0.01s	1	13	20
	Critical Section	0.01s	5	27	46
	Kanban	?	≥ 20	?	?
Tree	Percolate	0.05s	2	34	933
	Tree Arbiter	0.7s	2	88	7680
	Leader Election	0.1s	2	74	362
Ring	Token Passing	0.01s	2	2	2

We report the results in Table 1, running on a 2.4 GHz laptop with 4GB memory. We have categorized the experiments per topology. We display the running times (in seconds), the value of k and the final number of views generated ($|V|$). In most cases, the method terminates almost immediately illustrating the *small model property*: all patterns occur for small instances of the system. Observe that the sizes of the views are small as well, confirming the intuition that interactions between processes are of limited scope.

The bulk of the algorithm lies in the computation of the set $\gamma_k^{k+\ell}(V)$ and also the set $\mathcal{R}_k$. An example on which the algorithm fails is the *Kanban* system from [24]. This is a typical case where the cut-off condition is satisfied at high values of k. [24] refers to the computation of, at least, the set $\mathcal{R}_{20}$. $\mathcal{R}_{20}$ is large and so is the concretization of its views.

7 Related Work

An extensive amount of work has been devoted to regular model checking, e.g. [25,12]; and in particular augmenting regular model checking with techniques such as widening [9,32], abstraction [10], and acceleration [5]. All these works rely on computing the transitive closure of transducers or on iterating them on regular languages. Our method is significantly simpler and more efficient.

A technique of particular interest for parameterized systems is that of *counter abstraction*. The idea is to keep track of the number of processes which satisfy a certain property [22,17,13,14,30]. In general, counter abstraction is designed for systems with unstructured or clique architectures. As mentioned, our method can cope with these kinds of systems but also with more general classes of topologies. Several works reduce parameterized verification to the verification of finite-state models. Among these, the *invisible invariants* method [6,31] and the work of [29] exploit cut-off properties to check invariants for mutual exclusion protocols. The success of the method depends on the heuristic used in the generation of the candidate invariant. This method sometimes (e.g. for German's protocol) requires insertion of auxiliary program variables for completing the proof. The nature of invariants generated by our method is similar to that of the aforementioned works, since our invariant sets of views of size at most k can be seen as universally quantified assertions over reachable k-tuples of processes.

In [7], finite-state abstractions for verification of systems specified in WS1S are computed on-the-fly by using the weakest precondition operator. The method requires the user to provide a set of predicates on which to compute the abstract model.

The idea of refining the view abstraction by increasing k is similar in spirit to the work of [28] which discusses increasing precision of thread modular verification (Cartesian abstraction) by remembering some relationships between states of processes. Their refinement mechanism is more local, targeting the source of undesirable imprecision; however, it is not directly applicable to parameterized verification.

Environment abstraction [11] combines predicate abstraction with the counter abstraction. The technique is applied to Szymanski's algorithm. The model of [11] contains a more restricted form of global conditions than ours, and also does not include

features such as broadcast communication, rendez-vous communication, and dynamic creation and deletion of processes.

Recently, we have introduced the method of *monotonic abstraction* [3] that combines regular model checking with abstraction in order to produce systems that have monotonic behaviors w.r.t. a well quasi-ordering on the state space. In contrast to the method of this paper, the abstract system still needs to be analyzed using full symbolic reachability analysis on an infinite-state system. The only work we are aware of which attempts to automatically verify systems with non-atomic global transitions is [4] which applies monotonic abstraction. The abstraction in this case amounts to a verification procedure that operates on unbounded graphs, and thus is a non-trivial extension of the existing framework. As we saw, our method is easily extended to the case of non-atomic transitions.

The method of [21,20] and its reformulated, generic version of [19] are in principle similar to ours. They come with a complete method for well-quasi ordered systems which is an alternative to backward reachability analysis based on a forward exploration. Unlike our method, they target well-quasi ordered systems only and have not been instantiated for topologies other than multisets and lossy channel systems.

Constant-size cut-offs have been defined for ring networks in [16] where communication is only allowed through token passing. More general communication mechanisms such as guards over local and shared variables are described in [15]. However, the cut-offs are linear in the number of states of the components, which makes the verification task intractable on most of our examples.

The closest work to ours is the one in [24] that also relies on dynamic detection of cut-off points. The class of systems considered in [24] corresponds essentially to Petri nets. In particular, it cannot deal with systems with linear or tree-like topologies. The method relies on the ability to perform backward reachability analysis on the underlying transition system. This means that the algorithm of [24] cannot be applied on systems with undecidable reachability problems (such as the ones we consider in this paper). The method of [24] is yet complete.

8 Conclusion and Future Work

We have presented a uniform framework for automatic verification of different classes of parameterized systems with topologies such as words, trees, rings, or multisets, with an extension to handle non-atomic global conditions. The framework allows to perform parameterized verification by only considering a small set of instances of the system. We have proved that the presented algorithm is complete for a wide class of well quasi-ordered systems. Based on the method, we have implemented a prototype which performs efficiently on a wide range of benchmarks.

We are currently working on extending the framework to the case of multi-threaded programs operating on dynamic heap structures. These systems have notoriously complicated behaviors. Showing that verification can be carried out through the analysis of only a small number of threads would allow for more efficient algorithms for these systems. Furthermore, our algorithm relies on a very simple abstraction function, where a configuration of the system is approximated by its sub-structures (subwords, subtrees,

etc.). We believe that our approach can be lifted to more general classes of abstractions. This would allow for abstraction schemes that are more precise than existing ones, e.g., thread-modular abstraction [18] and Cartesian abstraction [27].

Obviously, the bottleneck in the application of the method is when the cut-off condition is only satisfied at high values of k (see e.g., the *Kanban* example in Section 6). We plan therefore to integrate the method with advanced tools that can perform efficient forward reachability analysis, like SPIN [23], and to use efficient symbolic encodings for compact representations for the set of views.

Acknowledgements. This work was supported by the Uppsala Programming for Multicore Architectures Research Center (UpMarc) and the Czech Science Foundation (project P103/10/0306).

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bulletin of Symbolic Logic 16(4), 457–515 (2010)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321 (1996)
3. Abdulla, P.A., Delzanno, G., Ben Henda, N., Rezine, A.: Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
4. Abdulla, P.A., Ben Henda, N., Delzanno, G., Rezine, A.: Handling Parameterized Systems with Non-atomic Global Conditions. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 22–36. Springer, Heidelberg (2008)
5. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular Model Checking Made Simple and Efficient. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 116–130. Springer, Heidelberg (2002)
6. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized Verification with Automatically Computed Inductive Assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
7. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
8. 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)
9. Boigelot, B., Legay, A., Wolper, P.: Iterating Transducers in the Large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
10. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
11. Clarke, E., Talupur, M., Veith, H.: Environment Abstraction for Parameterized Verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006)
12. Dams, D., Lakhnech, Y., Steffen, M.: Iterating Transducers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 286–297. Springer, Heidelberg (2001)
13. Delzanno, G.: Automatic Verification of Cache Coherence Protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)

14. Delzanno, G.: Verification of consistency protocols via infinite-state symbolic model checking. In: FORTE 2000. IFIP Conference Proceedings, vol. 183, pp. 171–186. Kluwer (2000)
15. 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)
16. Emerson, E.A., Namjoshi, K.: Reasoning about rings. In: POPL 1995, pp. 85–94 (1995)
17. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999. IEEE Computer Society (1999)
18. 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)
19. Ganty, P., Raskin, J.-F., Van Begin, L.: A Complete Abstract Interpretation Framework for Coverability Properties of WSTS. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg (2006)
20. Geeraerts, G., Raskin, J.-F., Van Begin, L.: Expand, Enlarge and Check.. Made Efficient. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 394–407. Springer, Heidelberg (2005)
21. Geeraerts, G., Raskin, J.F., Begin, L.V.: Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. J. Comput. Syst. Sci. 72(1), 180–203 (2006)
22. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
23. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
24. Kaiser, A., Kroening, D., Wahl, T.: Dynamic Cutoff Detection in Parameterized Concurrent Programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)
25. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theor. Comput. Sci. 256, 93–112 (2001)
26. Lynch, N.A., Shamir, B.P.: Distributed algorithms, lecture notes for 6.852, fall 1992. Tech. Rep. MIT/LCS/RSS-20, MIT (1993)
27. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-Modular Verification Is Cartesian Abstract Interpretation. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 183–197. Springer, Heidelberg (2006)
28. Malkis, A., Podelski, A., Rybalchenko, A.: Precise Thread-Modular Verification. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 218–232. Springer, Heidelberg (2007)
29. 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)
30. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0,1,\infty)$-Counter Abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
31. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic Deductive Verification with Invisible Invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
32. Touili, T.: Regular Model Checking using Widening Techniques. Electronic Notes in Theoretical Computer Science 50(4) (2001); Proc. of VEPAS 2001

Uncovering Symmetries in Irregular Process Networks

Kedar S. Namjoshi[1] and Richard J. Trefler[2,⋆]

[1] Bell Laboratories, Alcatel-Lucent
kedar@research.bell-labs.com
[2] University of Waterloo
trefler@cs.uwaterloo.ca

Abstract. In this work, we consider distributed protocols that operate on arbitrary networks. The analysis of such protocols is challenging, as an arbitrarily chosen network may have limited global symmetry. We describe a methodology that uncovers significant *local* symmetries by appropriately abstracting node neighborhoods in a network. The local symmetries give rise to uniform compositional proofs of correctness. As an illustration of these ideas, we show how to obtain a uniform compositional invariance proof for a Dining Philosophers protocol operating on a fixed-size, arbitrary network. An interesting and somewhat unexpected consequence is that this proof generalizes easily to a parametric proof, which holds on any network regardless of size or structure.

1 Introduction

A distributed protocol may be viewed as a collection of processes communicating over an underlying interconnection network. In many protocols, the processes are similar, while the network may be arbitrary. Examples are networking protocols such as TCP/IP and BGP, application-level protocols such as termination detection and global snapshot, and protocols for sensor and ad-hoc networks. The verification questions are (1) the analysis of arbitrary, fixed-size instances and (2) showing correctness in the parameterized sense; i.e., over an unbounded set of network instances.

These analysis questions are challenging, in large part because standard symmetry arguments do not apply to networks with irregular structure. On the other hand, proofs carried out by hand (e.g., those in [3]) make few distinctions between nodes; the typical inductive invariant has the uniform shape "for every node m, ...". This observation motivates our work. We conjecture that many distributed protocols can be analyzed uniformly, even if the underlying networks are irregular. Furthermore, we also conjecture that, once discovered, the uniformity can guide the construction of a parameterized proof. The parameterized model checking question is undecidable in general [2].

⋆ Supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 496–514, 2013.

To make progress on this conjecture, we look to a combination of abstraction and compositional reasoning. The components of our analysis are as follows.

1. Uncover local similarities in a network by abstracting node neighborhoods.
2. Perform a compositional analysis on the abstracted network to fully exploit the newly uncovered local symmetries; the result is an inductive invariant.
3. Check whether the reductions due to local symmetries are powerful enough for the invariant to be parametric.

Compositional analysis, by its nature, is less sensitive to global irregularities in network structure. This is because the analysis is carried out for each node individually, taking into account interference only from neighboring nodes. In recent work [19], we showed that the limited sensitivity makes it possible for compositional methods to take advantage of local symmetries in a network. As an example, consider a ring network of N nodes. The global symmetry group of the ring has size $O(N)$. Hence, standard symmetry reduction methods have limited effect: a state space of size potentially exponential in N can be reduced only by a linear factor. On the other hand, any two nodes are locally similar, as their immediate neighborhoods are identical. Using this local symmetry, a compositional invariant can be computed on a *single* representative node. This reduction also enables a parametric proof, as the representative may be chosen to be the same for all ring networks.

These earlier results, however, find their best application to networks with a regular structure, such as star, ring, mesh, and complete networks. In an irregular network, two obstacles arise. The first is that nodes may have different numbers of neighbors; this suffices to make them locally dissimilar. Even if all nodes have the same degree, irregular connectivity may limit the degree of *recursive* local similarity, called "balance", which is needed for the most effective symmetry reduction. To obtain a uniform analysis for irregular networks, it is necessary, therefore, to redefine local symmetry in a more general form. We do so in this work, which makes the following contributions.

- We formulate a notion of *local symmetry up to abstraction*. This generalizes the structural definition of local symmetry from [19] to a semantic one.
- We show that nodes that are "balanced" (i.e., recursively locally similar) have similar components in the strongest compositional invariant.
- Hence, an compositional invariant can be calculated using only a single representative from each equivalence class of balanced nodes.
- We show completeness: for any compositional invariant, it is always possible to derive it through a network abstraction based on a small set of local predicates, one that creates a highly locally-symmetric abstract network.
- We illustrate these ideas by showing how local symmetries may be used to calculate a parametric invariant for a Dining Philosophers protocol.

2 Abstraction Uncovers Symmetry

It is well understood that standard symmetry reduction [4,12] is a form of abstraction: symmetric states are bisimular, and the reduction abstracts a state to its bisimulation equivalence class. This work illustrates a converse principle: that abstraction may help uncover hidden symmetries. We demonstrate this with an example based on global symmetries. The subsequent sections work out this principle for local symmetries.

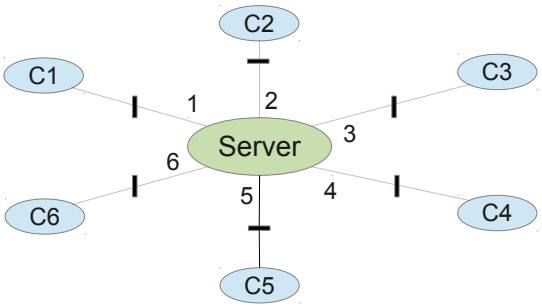


Fig. 1. A client-server network

The example is a client-server protocol with N identical clients. A 6-client instance is shown in Figure 1. The server controls access to a shared resource. Each client may request access to the resource, in which case it waits for a grant and eventually returns the resource to the server. To ensure fairness, the server cycles through its adjacent edges in clockwise order: if the last grant was given to the client on edge number c and the resource is available, the next grant is given to the first requesting client encountered when examining edges in clockwise order – i.e., in the sequence $c+1, c+2, ..., c+N$, where addition is modulo N.

This system has an exponential reachable state space, of size at least 2^N, as the subset of clients that have an outstanding request could be arbitrary. Although the picture suggests that any pair of clients can be interchanged, the operation of the server restricts the group of isomorphisms of the system to that of a ring. Hence, the degree of reduction that is possible is only $O(N)$.

An important safety property is mutual exclusion: at any time, at most one client should have a grant. This can be established with a simpler server process: if the resource is available, the server chooses a requesting client nondeterministically and grants its request. Formally, the abstract server simulates the original; hence, the abstract system as a whole simulates the original system. Moreover, the abstract system satisfies mutual exclusion. The abstract system has an exponential number of states as well. However, its automorphism group is the full symmetry group on the clients, so its state space can be reduced by an exponential factor.

3 Background: System Model, Compositional Invariants

This section introduces the system model, compositional invariants, and the fixpoint computation which produces the strongest such invariant. The material is largely a summary of [18,7,19].

System Model. A network is given as a graph. The graph has nodes and edges as objects, which are each given a color. Every node is connected to a set of edges; this set is ordered at the node according to an arbitrarily defined numbering. With respect to a node, a connected edge is either an input edge, an output edge, or both an input and an output edge. The network shown in Figure 1 has this form; edges are represented by rectangles and the numbering of edges on the server side is shown. Each edge is both an input and an output edge for its adjacent client and server node.

An *assignment* maps a set of processes to a network. The assignment also defines a state type for each node and each edge; types correspond to the coloring of the nodes and edges. The process assigned to a node m is denoted P_m. The *internal state* of this process is given by a value of the node state type. A *local state* of this process is given by a tuple of the form $(i, v_1, v_2, \ldots, v_k)$, where i is an internal state, and $v_1, v_2, \ldots, v_k$ is a valuation to the states of its adjacent edges, ordered by the numbering assigned to the edges at node m. For convenience, we associate symbolic variable names to all nodes and edges. The set of all node and edge variables is denoted V. The set of variables for process P_m is denoted V_m; these variables represent its internal state and the state of its adjacent edges. Hence, a local state is a valuation to V_m. The transition relation for P_m is denoted T_m. It relates local states of m with the constraint that if (s, t) is in T_m then the values for non-output edges of node m must be identical in s and t.

A *global* state is a valuation to all variables. Equivalently, a global state can be viewed as a set of local states where the local states for any two nodes agree on the value assigned to any common edge. The set of *global initial states* is denoted I. The projection of I on m, the set of *local initial states* for P_m, is denoted by I_m. Symbolically, I_m may be written as $(\exists V \backslash V_m : I)$. The quantification over all variables not in V_m projects I onto V_m. The global transition graph is induced by interleaving transitions from individual processes, starting from an initial state. There is a transition by process m from global state s to global state t if $T_m(s[V_m], t[V_m])$ holds ($s[V_m]$ is the local state of m in s) and for every variable x not in V_m, $s[x] = t[x]$.

Inductive and Compositional Invariants. An *inductive invariant* for a transition system is a set of global states which (a) includes all initial states and (b) is closed under all transitions. Formally, for an invariant ξ, condition (a) is denoted as $[I \Rightarrow \xi]$ and condition (b) as $[SP(T_i, \xi) \Rightarrow \xi]$, for all i. SP is the strongest post-condition operator (also known as the successor function, or as *post*).

A *compositional invariant* (called a "split" invariant in [18]) is an inductive invariant of a special shape: it is formed from the conjunction of a number of

local invariants, one for each process. Hence, it can be represented as a vector, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$, where each θ_i is defined over V_i and is itself an inductive invariant for process P_i. Equivalently, the constraints defining a compositional invariant are as follows.

- (Initiality) θ_i includes all initial states of P_i; formally, $[I_i \Rightarrow \theta_i]$
- (Step) θ_i is closed under transitions of P_i; formally, $[SP_i(T_i, \theta_i) \Rightarrow \theta_i]$, and
- (Non-interference) θ_i is closed under actions of neighboring processes. Formally, $[SP_i(intf^{\theta}_{ki}, \theta_i) \Rightarrow \theta_i]$, for any process k which points to i.

The predicate transformer SP_i is the strongest post-condition operator for node i. Node k *points to* node i if an output edge of k is adjacent to i. A transition of k may modify a local state of node i only if k points to i.

The term $intf^{\theta}_{ki}$ (read as "interference by k on i") is a transition condition. It describes how the local state of m may be changed due to moves by process k from states in its local invariant. Formally, $intf^{\theta}_{ki}$ is the projection of T_k, under θ_k, on to variables shared with m. This can be written symbolically as $intf^{\theta}_{ki} = (\exists V \backslash V_i, V' \backslash V'_i : T_k \wedge \theta_k)$, where the primed variables denote next-state values.

The Strongest Compositional Invariant as a Least Fixpoint. Grouping together the initiality, step, and non-interference constraints gives a set of simultaneous implications of the shape $[F_i(\theta) \Rightarrow \theta_i]$. Here, $F_i(\theta)$ is the disjunction of the terms appearing on the left-hand side of the constraints for θ_i: namely, I_i, $SP_i(T_i, \theta_i)$, and $SP_i(intf^{\theta}_{ki}, \theta_i)$ for all k pointing to i. As F_i is monotonic in θ for all i (vectors are ordered by point-wise implication), the set of constraints has a least vector solution by the Knaster-Tarski theorem. The least solution, denoted by θ^*, forms the strongest solution to the constraints, and is therefore the strongest compositional invariant.

The least fixpoint is calculated in the standard manner by a process of successive approximation. The initial approximation, θ^0_i, is the empty set for all i. The approximation $\theta^{(K+1)}_i$ for stage $(K+1)$ is defined as $F_i(\theta^K)$. Standard methods, such as widening, may be used to ensure convergence for infinite-state systems. This is a synchronized computation. However, by the chaotic iteration theorem of [9], the simultaneous least fixpoint may be computed in an asynchronous manner, following any "fair" schedule (one in which each component is eventually given a turn). In Figure 2 we show one possible implementation of the computation.

```
var θ, θ′: vector
initially, for all i: θ_i = ∅, θ′_i = I_i
while (θ ≠ θ′) do
   forall i: θ_i := θ′_i
   forall i: θ′_i := θ_i ∨ SP_i(T_i, θ_i) ∨ ( ∨ k : k points-to i : SP_i(intf^θ_ki, θ_i))
done
```

Fig. 2. Computing the Compositional Fixpoint

4 Informal Analysis of a Dining Philosophers Protocol

In this section, we describe a protocol for the Dining Philosophers problem and outline an analysis which performs local abstraction to extract symmetry. This is done in an informal manner; the justification for the soundness of these steps is laid out in the following sections.

The Protocol. We model a Dining Philosophers protocol (abbreviated by DP) as follows. The protocol consists of a number of similar processes operating on an arbitrary network. Every edge on the network models a shared "fork"; the variable for the edge between nodes i and j is called f_{ij}. Its domain is $\{i, j, \perp\}$. Node i is said to *own* the fork f_{ij} if $f_{ij} = i$; node j owns this fork if $f_{ij} = j$; and the fork is available if $f_{ij} = \perp$.

The process at node i goes through the following internal states: T (thinking); H (hungry); E (eating); and R (release). Each state s is really a "super-state" with a sub-state s_X for every subset X of adjacent forks, but we omit this detail for simplicity. Let nbr be the neighbor relation between processes. The state transitions for a process are as follows.

- A transition from T to H is always enabled.
- In state H, the process acquires forks, but may also choose to release them
  - (acquire fork) if $nbr(i, j)$ and $f_{ij} = \perp$, set $f_{ij} := i$,
  - (release fork) if $nbr(i, j)$ and $f_{ij} = i$, set $f_{ij} := \perp$, and
  - (to-eat) if $(\forall j : nbr(i, j) : f_{ij} = i)$ holds, change state to E.
- A transition from E to R is always enabled.
- In state R, the process releases its owned forks.
  - (release fork) if $nbr(i, j)$ and $f_{ij} = i$, set $f_{ij} := \perp$
  - (to-think) if $(\forall j : nbr(i, j) : f_{ij} \neq i)$, change state to T

The initial state of the system is one where all processes are in internal state $T_\emptyset$ and all forks are available (i.e., have value $\perp$).

Correctness Properties. The desired safety property is that there is no reachable global state where two neighboring processes are in the eating state (E). The protocol given above is safe. It is also free of deadlock, as a process may always release a fork to its neighbor. It is, however, not free of livelock. Our focus is on a proof of the safety property of mutual exclusion between neighbors.

Abstract DP Model. The simplest abstraction is to have just the four abstract states: T, H, E, R, corresponding to the four super-states. The abstract transitions derived from standard existential abstraction are $T \to H, H \to H, H \to E, E \to R, R \to R, R \to T$. However, this is too coarse an abstraction for compositional analysis. By the Step rule (Section 3) all four states, being reachable from the initial abstract state T, must belong to the final invariant, θ_i^*, for every process i. This abstract compositional invariant contains a global state where

neighbors i, j are in state E, which violates the desired property of mutual exclusion between neighbors.

To tighten up the abstraction, we define a predicate A that is true for a node i if it "owns all adjacent forks" (i.e., if for every j adjacent to i, the fork variable on the edge (i, j) has value i). Note that this predicate occurs in the protocol, guarding the transition from state H to state E. The reachable abstract transitions at a node with at least one adjacent edge are shown in Figure 3(a). For an isolated node A is vacuously true as it has no adjacent forks; the transitions for such a node are shown in Figure 3(b).

The standard existential abstraction is used to compute these transitions. The concrete domain for node m is the set of local states of m, L_m. The abstract domain is the set $\{T, H, E, R\} \times \{\mathsf{A}, -\mathsf{A}\}$. The abstraction function, α_m, maps a local state s to an abstract state based on the super-state in s and the value of A in s. This induces a Galois connection (α_m, γ_m) connecting the two domains. There is an abstract transition from (abstract) state a to (abstract) state b if there exist local states x, y of node m such that $x \in \gamma_m(a)$, $y \in \gamma_m(b)$, and $T_m(x, y)$ holds.

Abstract Interference Transitions. Figure 3 shows the abstract states reachable through step transitions. For the compositional analysis, we also have to consider how interference by nodes adjacent to node m affects the abstract states of node m. The concrete interference due to node k was defined in Section 3. Expanding the definition of $intf^{\theta}_{km}$, one gets that (u, v) is an interference transition for node m caused by node k, under a vector of assertions θ, if the following holds.

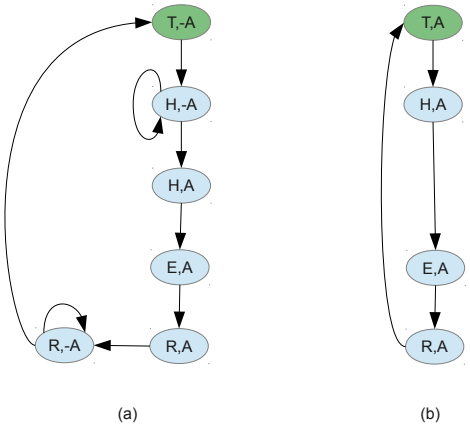


Fig. 3. Abstract State Transitions (a) for non-isolated nodes and (b) for an isolated node. The notation "$-\mathsf{A}$" indicates the negation of A. Green/dark states are initial.

$$(\exists s, t : u = s[V_m] \ \wedge \ v = t[V_m] \ \wedge \ T_k(s,t) \ \wedge \ \theta_k(s[V_k])) \tag{1}$$

Here, states s, t are *joint* states of nodes m and k, representing an assignment of values to the variables in $V_m \cup V_k$.

By analogy, the interference of node k on the abstract state of m is defined as follows. An abstract transition (a, b) for node m is the result of interference by process P_k under a vector of *abstract* assertions ξ if the following holds.

$$(\exists s, t : a = \alpha_m(s[V_m]) \ \wedge \ b = \alpha_m(t[V_m]) \ \wedge \ T_k(s,t) \ \wedge \ \xi_k(\alpha_k(s[V_k]))) \tag{2}$$

Informally, this expression says that every abstract interference transition is witnessed by a concrete interference transition.

Computing Interference. Surprisingly, *there is no non-trivial interference in the abstract DP protocol*! Informally, this is due to the following reason. The predicate A_m refers to the set of forks owned by m. In the concrete protocol, an adjacent node cannot take away ownership of forks from node m, nor can it grant ownership of forks to m. Hence, the value of A_m is unchanged by transitions at neighboring nodes. Those transitions cannot change the value of the internal state of node m, either. Formally, the maximum interference from k on to m, obtained by setting ξ_k to *true* in the defining expression (2), shows that the abstract state of m is unchanged by transitions of k.

Abstract Compositional Invariants. We have just established that in the abstract domain, all interference is trivial. Hence, the compositional invariance calculation produces, for each process, only the set of states reachable via abstract step transitions. This is just the set of states shown in Figure 3. From it, one can read off the following invariant: for all nodes m, if E_m is true, then A_m is true. The corresponding concrete invariant is that for all nodes m, if E_m is true, then $\gamma_m(\mathsf{A}_m)$ is true. There cannot be a global state meeting this invariant where adjacent nodes m and n are each in state E. Otherwise, $\gamma_m(\mathsf{A}_m)$ and $\gamma_n(\mathsf{A}_n)$ will be true simultaneously, which is impossible – recall that A_m states that m owns *all* adjacent forks. Hence, the invariant suffices to show exclusion between neighbors.

Symmetry Reduction. The abstract transition graphs are identical for all non-isolated nodes, and identical for all isolated nodes. It suffices, therefore, to have a single representative node for each class. Hence, the analysis of an arbitrary network, however irregular, can be reduced to the analysis (under abstraction) of *two* representative nodes. As this holds for all networks, the (abstract) compositional invariant calculated for a small representative network, one with each kind of node, defines a *parameterized proof* of safety.

Next Steps. In the following sections, we build up the foundations required to show the soundness of this informal analysis. We give a definition of abstract

symmetry, specialize it to the case of predicate abstraction, and show the consequences for symmetry reduction. We also show a completeness result generalizing the observation made for DP that its abstract transition graph is interference-free. We show that there is always an abstraction for which this is true if there exists a parameterized conjunctive invariant for the protocol.

A similar analysis to the one carried out here applies also to another Dining Philosophers protocol where there is always a distinguished node. In the case of a ring network, the process at the distinguished node, say P_0, chooses its forks in an order that is the reverse of that taken by other processes, for instance, in the order left;right instead of right;left. (This ensures the absence of deadlock.) The irregularity introduced by the distinguished process implies that the only structural balance relation for this ring is the trivial identity relation. However,in the semantic balance relation after abstraction, any two nodes are equivalent.

5 Local Similarity Up to Abstraction

In this section, we develop the theory combining local abstraction and symmetry. In [19], we introduced the notion of a local similarity between nodes of a network. We refer to that as *structural* similarity here, to distinguish it from the semantic similarity notion to follow.

5.1 Structural Similarity and Balance

Two nodes in a network are structurally similar if they have the same color and there is a bijection between their neighborhoods. The neighborhood of a node is the set of its adjacent edges. The bijection should respect the edge color and the type of the node-edge connection. I.e., input edges should be mapped to input edges and output edges to output edges. The similarity between nodes is represented by a triple, (m, β, n), where m, n are nodes and β is the witnessing bijection. The set of all similarity triples forms a *groupoid*, a group-like structure with a partial composition operation.

A *structural balance* relation is a subset of this groupoid that induces a recursive similarity. The definition has a co-inductive form, rather like that of a bisimulation. If (m, β, n) is in the balance relation, then for every node k that points to m, there is a node l that points to n and a map δ, such that (k, δ, l) is in the balance relation, and β and δ agree on the edges common to m and k. We have the following theorem.

Theorem 1 ([19]). *Consider a structural balance relation B. For any program that is a valid assignment relative to B, the strongest compositional invariant θ^* is such that for any (m, β, n) in B, $[\theta^*_n \equiv \beta(\theta^*_m)]$.*

A program is a valid assignment relative to B if the assignment respects the symmetries in B: if (m, β, n) is in B, the transition relations and initial conditions for P_m and P_n must be related by β. The conclusion of the theorem says that for any state x in θ^*_m, the state y obtained from x by permuting edges according to β

belongs to θ_n^*. The permutation is given by setting $y(\beta(e)) = x(e)$ for every edge variable e. Informally, this theorem says that the strongest local invariants for any pair of structurally similar nodes are themselves similar. Hence, it suffices to compute θ^* for a representative from each class of balance-equivalent nodes, as shown in [19].

5.2 Semantic Similarity and Balance

Structural balance is defined solely on the network structure. This can be limiting, as irregular networks have only trivial structural balance relations. The semantic notion of balance mixes together program and network structure. It requires balanced nodes to have similar transition relations and similar interference from other nodes.

The analysis in Section 4 relies on abstracting the local state of each node. After this is done one cannot, in general, define interference between nodes in terms of shared state. Therefore, it is necessary to abstract the definition of interference. It is convenient to think of "interference" as a primitive notion: for node k that points to m, there is a postulated interference relation $intf_{km}^X$ that is a transition relation on the states of m, parameterized by a set X of states of k. This relation is used as usual in the fixpoint computation (Figure 2). We require that $intf_{km}^X$ is monotonic in X to ensure that the least fixpoint is defined.

A *semantic balance* relation consists of triples (m, β, n) where m and n are nodes and β is a relation on the local state sets L_m and L_n. (Recall that in a structural balance relation, β is a bijection on edges.) As with structural balance, there is a clause that propagates local symmetry between m and n to corresponding neighbors. In the following, we use the notation $\langle\beta\rangle Y$ for the set $\{x \mid (\exists y : (x, y) \in \beta \ \wedge \ y \in Y)\}$ of predecessors of Y according to β.

Definition 1. ***(Semantic One-sided Balance)*** *A one-sided balance relation is a set of triples such that for every (m, β, n) in the balance relation*

1. *(initial-similarity) initial states are related by β: formally, $[I_m \ \Rightarrow \ \langle\beta\rangle I_n]$*
2. *(step-similarity) β is a safety-simulation from T_m to T_n*
3. *(interference-similarity) For every k that points to m, there is l that points to n and δ for which*
   (a) *(successive-balance) (k, δ, l) is in the balance relation, and*
   (b) *(agreement) For state sets X, Y such that $[X \ \Rightarrow \ \langle\delta\rangle Y]$, β is a safety-simulation from $intf_{km}^X$ to $intf_{ln}^Y$.*

A safety simulation (cf. [21]) R from T_1 to T_2 is defined as follows. For any $(s, t) \in R$ and any transition $(s, s') \in T_1$, there must be state t' such that $(t, t') \in T_2^*$ and $(s', t') \in R$. It is a form of simulation, weakened by the use of the reflexive transitive closure T_2^*.

Definition 2. ***(Two-sided Balance)*** *A two-sided balance relation is a one-sided balance relation that is closed under inverse; i.e., if (m, β, n) is in the relation, so is (n, β^{-1}, m).*

Theorem 2. *Every structural balance relation together with a valid assignment induces a two-sided balance relation.*

5.3 Symmetry Reduction

We consider the compositional fixpoint computation to be carried out in stages. The set of local states computed for node m at stage S is denoted θ_m^S. At the initial stage ($S = 0$), $\theta_m^S = I_m$. In a "step" stage, step transitions are applied for all nodes until no new states are generated. In an "interference" stage, interference transitions are applied for all nodes until no new states are generated. The main symmetry theorem below shows that for a balance triple (m, β, n), at every stage, the local states generated at m are related through β to the local states generated by n.

Theorem 3. *(Main Symmetry Theorem) Given a semantic balance relation, at every fixpoint stage S in the compositional invariance calculation, for all (m, β, n) in the balance relation, $[\theta_m^S \Rightarrow \langle\beta\rangle\theta_n^S]$ holds.*

Proof. The proof is by induction on stages.

Consider the initial stage. Here, the θ-values are the initial state sets for each process. The claim follows from the initial-similarity condition.

Suppose, inductively, that the claim holds for stage S for all triples in the balance relation. Consider stage $S+1$ and the triple (m, β, n). For node m, let s be a state generated at stage $S+1$ that is an immediate successor of a state u in θ_m^S. From the induction hypothesis applied to u, there is a state v in θ_n^S related by β to u. There are two cases, based on the type of stage $S+1$.

(1) Suppose that $S+1$ is a step stage, so that s is a successor by T_m. By step-similarity, there is a state t reachable by a sequence of T_n moves from v that is β-related to s. As the stage $(S+1)$ calculation closes-off under step successors, t belongs to θ_n^{S+1}.

(2) Suppose, instead, that $S+1$ is an interference stage, and that s arises from an interference transition by node k that points to m. By balance, there is l that points to n and δ such that (k, δ, l) is in the balance relation.

Let $X = \theta_k^S$ and $Y = \theta_l^S$. By the inductive claim applied to k and l, $[X \Rightarrow \langle\delta\rangle Y]$. From the agreement condition, β is a safety-simulation between $intf_{km}^X$ and $intf_{ln}^Y$. Thus, for the transition from u to s, which is in $intf_{km}^X$, there is a matching sequence of $intf_{ln}^Y$-transitions from v to a state t that is related by β to s. As the stage $(S+1)$ calculation closes-off under interference successors, t belongs to θ_n^{S+1}.

We have considered only the important case, where s is an immediate successor of a state in the previous stage. The case of a non-immediate successor within the same stage may be shown by induction within a stage based on the length of its derivation path within that stage. □

Corollary 1. *Consider a two-sided semantic balance relation where every triple is based on a one-to-one relation. For any (m, β, n) in the balance relation and every fixpoint stage S, $[\theta^S_m \equiv \langle\beta\rangle\theta^S_n]$.*

Proof. The direction from left-to-right follows from Theorem 3. As the balance relation is two-sided, it includes the triple (n, β^{-1}, m). By Theorem 3 for this triple, $[\theta^S_n \Rightarrow \langle\beta^{-1}\rangle\theta^S_m]$. Hence, $[\langle\beta\rangle\theta^S_n \Rightarrow \langle\beta\rangle\langle\beta^{-1}\rangle\theta^S_m]$. As β is one-to-one, this implies that $[\langle\beta\rangle\theta^S_n \Rightarrow \theta^S_m]$. □

Based on Theorem 3 and Corollary 1, one may symmetry-reduce the fixpoint calculation as follows, using a procedure defined in [19]. Consider a balance relation that is closed under inverse and composition. This defines a symmetry groupoid. The orbit of the groupoid, i.e., the set of pairs (m, n) such that there is a β for which (m, β, n) is in the balance relation, is an equivalence relation. For each balance-equivalence class, one chooses a representative node. The fixpoint calculation is carried out only for the representative nodes. The value of θ^S_i for a non-representative node i, which may be needed to calculate interference, can be computed by Corollary 1 as $\langle\beta\rangle\theta^S_r$), where r is the representative for node i and (i, β, r) is a triple linking i to r.

6 Local Predicate Abstraction and Symmetry Reduction

This section connects the general concepts from Section 5 with the predicate abstractions used in the informal treatment in Section 4.

6.1 Local Domain Abstraction

We consider the effect of a general domain abstraction before specializing to predicate abstraction. The effect of a domain abstraction at each node is to construct an abstract network, which has the same structure as the original one, but with a different state space at each node. We refer to the abstract network for network N as $\overline{N}$ and refer to the abstract counterpart of node m as $\overline{m}$. In the following, we show how to connect the two networks using a balance relation. This relation induces a connection between the compositional invariants computed on the concrete and abstract networks.

A *local abstraction* is given by specifying, for each node m an abstract domain, D_m, and a total abstraction function, $\alpha_m : L_m \to D_m$. This induces a Galois connection on subsets, which we also refer to as (α_m, γ_m): $\alpha_m(X) = \{\alpha_m(x) \mid x \in X\}$, and $\gamma_m(A) = \{x \mid \alpha_m(x) \in A\}$.

The abstract set of initial states, $\overline{I}_m$ is given by $\alpha_m(I_m)$. (For a simpler notation, we denote this set as $\overline{I}_m$ rather than $I_{\overline{m}}$.) The abstract step transition, $\overline{T}_m$, is obtained by existential abstraction: there is a transition from (abstract) state a to (abstract) state b if there exist x, y in L_m such that $\alpha_m(x) = a$, $\alpha_m(y) = b$, and $T_m(x, y)$ holds. An abstract transition (a, b) is the result of

interference by node k from state set Y, that is, $(a,b) \in \overline{intf}^{Y}_{km}$, if the following holds.

$$(\exists s,t : \alpha_m(s[m]) = a \ \wedge \ \alpha_m(t[m]) = b \ \wedge \ T_k(s,t) \ \wedge \ \alpha_k(s[k]) \in Y) \qquad (3)$$

Theorem 4. *The set of triples of the form $(m, \alpha_m, \overline{m})$ is a one-sided semantic balance relation connecting the concrete network N to the abstract network $\overline{N}$.*

Proof. We have to check the conditions for one-sided balance. Initial-similarity follows by the definition of $\overline{I}_m$. Step-similarity holds as the abstract transition relation is the standard existential abstraction. For a node k that points to m, we use $\overline{k}$ as its corresponding node, which points to $\overline{m}$, and let $\delta = \alpha_k$. Then, $(k, \delta, \overline{k})$ is also in the balance relation. The agreement condition follows from the analogy between the concrete and abstract interference conditions. Specifically, if (x,y) is a transition in $intf^{X}_{km}$, there exist joint (concrete) states s,t such that $s[k] \in X$, $s[m] = x$, $t[m] = y$ and $T_k(s,t)$ all hold. Thus, we get that $\alpha_k(s[k]) \in \alpha_k(X)$, $\alpha_m(s[m]) = \alpha_m(x)$, $\alpha_m(t[m]) = \alpha_m(y)$ and $T_k(s,t)$ all hold. Let Y be such that $[X \ \Rightarrow \ \langle \alpha_k \rangle Y]$. Then we get that $\alpha_k(s[k]) \in Y$. Hence, from definition (3) above, the pair $(\alpha_m(x), \alpha_m(y))$ is in $\overline{intf}^{Y}_{km}$, which establishes the agreement condition. □

As a consequence, from Theorem 3, we obtain the following corollary. This corollary shows that the strongest compositional invariant obtained on $\overline{N}$ can be concretized to a compositional invariant for the network N. Abstraction may lose some precision, but this comes at the potential gain of local symmetry.

Corollary 2. *Let θ^* and ξ^* be the strongest compositional invariants for the concrete and abstract networks, respectively. Then (1) for every m, $[\theta^*_m \ \Rightarrow \ \gamma_m(\xi^*_{\overline{m}})]$ and (2) the vector $(m : \gamma_m(\xi^*_{\overline{m}}))$ is a compositional invariant for the concrete network.*

Proof (sketch). The first conclusion follows from Theorem 3 applied to the (disjoint) union of N and $\overline{N}$, and the Galois connection between α_m and γ_m.

The second conclusion follows by reasoning with the conditions of compositional invariance and the Galois connection. For simplicity, we suppose that the abstract domain is flat (i.e., ordering is equality), so that the condition $x \in \gamma_k(Y)$ is equivalent to $\alpha_k(x) \in Y$. (Non-flat domains may be handled by taking downward closures in the computation of ξ^*, a complication we omit here.)

Define $Z_m = \gamma_m(\xi^*_m)$ for all m. We show that Z_m satisfies the compositional invariance conditions for node m by considering them in turn. For $x \in I_m$, $\alpha_m(x)$ is in $\overline{I}_m$. By initiality, $\alpha_m(x)$ is in ξ^*_m, so that $x \in Z_m$. For $x \in SP_m(T_m, Z_m)$, there is $y \in Z_m$ such that $T_m(x,y)$. Hence, $(\alpha_m(x), \alpha_m(y))$ belongs to $\overline{T}_m$ and $\alpha_m(y)$ is in ξ^*_m. The step condition for $\overline{T}_m$ ensures that $\alpha_m(x)$ is in ξ^*_m; hence, $x \in Z_m$. Similar reasoning proves the case for interference transitions. □

6.2 Local Predicate Abstraction

Local predicate abstraction maps the local state of a node to the valuation of a set of predicates on the local state. For simplicity, we fix a set of predicates, $\mathcal{P}$, which can be interpreted over all nodes.

There is a natural Galois connection, denoted (α_m, γ_m), established by $\mathcal{P}$ over a node m. For a local state s of node m, the abstraction function $\alpha_m(s)$ maps s to a set of literals giving the valuation of the predicates in $\mathcal{P}$ on s. This is the set $\{(p \equiv p(s)) \mid p \in \mathcal{P}\}$. The abstraction is extended naturally to sets of concrete states. The concretization function, γ_m, maps a set X of of sets of literals to the set of local states given by $\{s \mid \alpha_m(s) \in X\}$. Given this abstraction, the abstract forms of the transition relation and interference are as defined in Section 6.1.

Corollary 2 establishes that the compositional invariant calculated using the abstract initial states, abstract step and abstract interference transitions defined over $\mathcal{P}$ is, as interpreted through the γ_m functions, a valid compositional invariant for the concrete system.

We now show that, with the right choice of predicates, the induced abstract network is (a) fully locally symmetric, (b) free of interference, and (c) adequate. This result is similar in spirit to a completeness theorem of Kesten and Pnueli [17] for abstraction over the global state space. The key difference in the following theorem is in its treatment of compositionality and symmetry, which are not covered by the Kesten-Pnueli result.

Theorem 5. *For a fixed-size process network, any inductive invariant of the form $(\forall i : \theta_i)$, where each θ_i is local to process P_i, can be established by compositional reasoning over a uniform abstract Boolean network.*

Proof (sketch). By a result in [18], the strongest compositional invariant, θ^*, is such that $[\theta_i^* \Rightarrow \theta_i]$, for all i.

The single predicate symbol is B. (A helpful mnemonic is to read B as "black" and $(\neg B)$ as "white".) The abstraction function α_i maps a local state x of L_i to B, if $\theta_i^*(x)$ holds, and to $(\neg B)$ otherwise. Let ξ^* be the strongest compositional invariant computed for the abstract network. By Corollary 2, $[\theta_i^* \Rightarrow \gamma_i(\xi_i^*)]$. This implies that the abstract state $(B = \mathit{true})$ is in ξ_i^* for all i.

We show that the implication is, in fact, an equality; i.e., that the other abstract state, $(B = \mathit{false})$, does not belong to ξ_i^* for any i. As $[I_m \Rightarrow \theta_m^*]$, it follows that $[\alpha_m(I_m) \Rightarrow \alpha_m(\theta_m^*)]$, i.e., that $[\overline{I}_m \Rightarrow B]$. Hence, initially, only the state B is in ξ_m^*. Suppose, inductively, that this is the case at the S'th stage. Consider an abstract step transition that introduces the state $(\neg B)$. This must have a concrete step transition $T_m(x, y)$ as a witness where $\theta_m^*(x)$ holds but $\theta_m^*(y)$ does not. This is impossible by the step constraint for θ^*. Similarly, one can establish the impossibility of an abstract interference transition of this kind, so that the only interference is the trivial (B, B) transition. Hence, $\xi_m^* = \{B\}$ for all m, so that the concrete invariant induced by ξ^* is just θ^*. This establishes adequacy: any property implied by the original invariant can be shown with the concretized abstract invariant.

We now show that, in the abstract network, any two isolated nodes and any two non-isolated nodes are balanced. The balance relation consists of triples (m, β, n) where β is the partial bijection $B \mapsto B$. Initial-similarity holds as the only initial state is B. Step similarity holds as, by the reasoning above, the only abstract reachable step transition is (B, B). The reasoning so far is sufficient to show that any two isolated nodes are balanced; the rest of the proof applies to the case where m and n are not isolated nodes. For a node k that points to m, choose its corresponding node l arbitrarily from the nodes pointing to n – there is at least one such node as n is not isolated. Then k and l are non-isolated and (k, δ, l) is in the balance relation with the bijection δ which maps B to B. Consider X, Y as in the agreement condition. It suffices to consider $Y = \delta(X)$, by the monotonicity of interference relations. As δ is defined for all elements of X it cannot include $(\neg B)$; therefore, X must be either $\emptyset$ or $\{B\}$. Thus, $Y = \delta(X)$ is correspondingly $\emptyset$ or $\{B\}$. Expanding the definition of abstract interference, it follows from the non-interference condition for θ^* that β is a safety simulation between $intf^X_{km}$ and $intf^Y_{ln}$. □

A consequence of Corollary 2 is that if all members of a parameterized family of networks can be reduced to a finite set of abstract networks (over a fixed set of predicates), the compositional invariants computed for the abstract networks concretize to a parametric compositional invariant for the entire family. The following theorem shows that there is always a "right" choice of predicate for which this is true.

Theorem 6. *For a parameterized family of process networks, any compositional invariant of the form* $(\forall i : \theta_i)$*, where each* θ_i *is local to process* P_i*, can be established by compositional reasoning over a small uniform abstract Boolean network.*

Proof (sketch). The difference between the statement of this theorem and Theorem 5 is that the assumed invariant is compositional. This implies that the compositionality conditions hold for the given θ_i for every instance. Hence, the predicate symbol B in the proof of Theorem 5 has an interpretation that is uniform across all instances.

The proof of Theorem 5 shows that in the abstract network, there are at most two classes of nodes: those that are isolated and those that are not. Hence, there is a cutoff instance N whose abstract network $\overline{N}$ (over B) exhibits all the classes of nodes that can arise. By Theorem 3, the concretized form of the compositional abstract invariant for $\overline{N}$ is a compositional invariant for every larger instance of the family. □

6.3 Reviewing the Dining Philosophers Analysis

We can now review the informal analysis of Section 4 in terms of these theorems. The per-node abstraction with predicate A is an instance of the local predicate abstraction discussed in Section 6.2. From Corollary 2, the abstract compositional invariant, when concretized, is a compositional invariant for the concrete

system. This establishes the correctness of $(\forall m : E_m \Rightarrow \gamma_m(\mathsf{A}_m))$ as a concrete invariant.

For the abstract network, the candidate balance relation is $\{(m, id, n)\}$ where id is the identity relation, and m, n are both isolated nodes or both non-isolated nodes. Using the definitions of abstract transitions and interference, one can check that this meets the conditions for a two-sided balance relation (Definitions 1 and 2). The orbit of this balance relation has just two classes so, from Corollary 1, it suffices to consider two representative nodes for the analysis. Since the calculation for the representative node is identical across networks, we may conclude that the computed invariant applies to all networks.

The completeness theorems show that the phenomenon observed in the informal analysis of the Dining Philosophers protocol (Section 4) is not an isolated case. Any compositional invariant of a parameterized family can be obtained through a local predicate abstraction that induces complete local symmetry and only trivial interference in the abstract network.

7 Summary and Related Work

The seminal work on symmetry reduction in model checking [12,4,16] and its many refinements base the theory on the *global* symmetries of a Kripke structure, expressed as a group of automorphisms. For a distributed program, these symmetries (as shown in [12]) are lower-bounded by the symmetries of the process interconnection network. In fact, for most interesting protocols, the symmetries are also upper-bounded by the group of symmetries of the process network. This implies that symmetry reduction works well for networks with a complete interconnection or a star shape. (This is typically a client-server structure, although, as the example in Section 2 shows, not all client-server protocols fit the assumptions made in [12].) For other networks, most notably ring, mesh and torus networks, the global automorphism group has size at most linear in the number of nodes of the network; hence, the available symmetry reduction is also at most a linear factor. This is not particularly helpful if, as is often the case, the Kripke structure is exponential in the size of the network.

Motivated by this problem, we introduced in [19] the notion of local symmetries. Regular networks, such as the complete, star, ring, mesh and torus networks have a high degree of local similarity: intuitively, the network "looks the same" from nearly every node. We show that this results in symmetry reduction for compositional methods. Although the compositional invariance calculation is polynomial, requiring time $O(N^3)$ for a network of N nodes, local symmetry does have a significant effect, in two ways. First, for many regular networks, the calculation time becomes independent of N after symmetry reduction. Second, it is possible to derive parametric invariants if local symmetry reduces each member of a family of networks to a fixed set of representatives.

As discussed in the introduction, this symmetry notion is, however, not applicable to irregular networks. In this work, we show that it is possible in many cases to overlay a local similarity structure on an irregular network, by using an

appropriate abstraction over node neighborhoods. The theoretical drawback is that using abstraction generally results in weaker invariants. On the other hand, we show that for the Dining Philosophers protocol, the invariant calculated by abstraction suffices to prove mutual exclusion. Moreover, the completeness result ensures that, for any compositional invariant, there is always an appropriate predicate abstraction. Hence, we conjecture that abstraction will suffice for most practical analysis problems.

Other compositional reasoning methods, such as those based on alternative assume-guarantee rules [15] or on automaton learning [14,6] should also benefit from local symmetry reduction; working out those connections is a subject for future work. It should be noted that there are other techniques (e.g., [13]) which enhance global symmetry in certain cases where the original protocol is only minimally globally symmetric. In the current work, we have instead applied local, rather than global, symmetry reduction techniques; local symmetries appear to be more widely applicable. A particularly intriguing outcome of the analysis of the Dining Philosophers protocol is that one can show a parametric invariant, one which holds for all networks. Parameterized safety analysis is undecidable in general [2]. There is a large variety of analysis methods, such as those based on well-quasi-orders (e.g., [1]) or on iterating transducers (e.g., [10]), each of which works well on a class of problems.

We discuss two methods which are the closest to the point of view taken here. The first is the "network grammar" method from [20]. A family of networks is described by a context-free network grammar. A choice of abstract process is made for each non-terminal in the grammar. This results in a set of model-checking constraints, which, if solvable, give a parametric proof of correctness. This technique is applicable to regular networks (rings, trees) that have a compact grammar description. The second method is that of environment abstraction [5]. This method chooses the point of view of a single process, abstracting the rest of the network. There is a certain similarity between the generic process used for environment abstraction and the single representative process used in our work. However, there is a difference in how the network abstraction is defined (non-compositionally for environment abstraction) and the method has not been applied to irregular networks.

The connections made in [19] between local symmetry, compositionality and parametric verification are extended in here to irregular networks. The crucial observation is that local abstraction can make an irregular network appear regular, facilitating symmetry reduction. The application to versions of the Dining Philosophers protocol and the completeness results suggest that these connections are worth further study. In ongoing work, we are examining how well abstraction works for other protocols. An interesting question is whether appropriate abstraction predicates, such as the predicate A from Section 4, can be discovered automatically. It is possible that automatic methods that discover auxiliary predicates to address incompleteness (e.g., [7,8]) can be adapted to discover predicates for abstraction. A particularly interesting question for future work is to investigate parametric proofs of protocols on dynamic networks, i.e.,

networks where links and nodes can fail or appear, a domain that is interesting because of its connections to fault tolerance and ad-hoc networking (cf. [11]).

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321. IEEE Computer Society (1996)
2. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
3. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988)
4. Clarke, E.M., Filkorn, T., Jha, S.: Exploiting Symmetry in Temporal Logic Model Checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 450–462. Springer, Heidelberg (1993)
5. Clarke, E., Talupur, M., Veith, H.: Environment Abstraction for Parameterized Verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006)
6. 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)
7. 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)
8. 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)
9. Cousot, P., Cousot, R.: Automatic synthesis of optimal invariant assertions: mathematical foundations. In: ACM Symposium on Artificial Intelligence & Programming Languages, Rochester, NY (August 1977); ACM SIGPLAN Not. 12(8), 1–12
10. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. J. Log. Algebr. Program 52-53, 109–127 (2002)
11. Delzanno, G., Sangnier, A., Zavattaro, G.: Verification of Ad Hoc Networks with Node and Communication Failures. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 235–250. Springer, Heidelberg (2012)
12. 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)
13. Emerson, E.A., Havlicek, J., Trefler, R.J.: Virtual symmetry reduction. In: LICS, pp. 121–131. IEEE Computer Society (2000)
14. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE, pp. 3–12. IEEE Computer Society (2002)
15. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344. ACM (2011)
16. Ip, C., Dill, D.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
17. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Information and Computation 163(1), 203–243 (2000)
18. 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)

19. Namjoshi, K.S., Trefler, R.J.: Local Symmetry and Compositional Verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 348–362. Springer, Heidelberg (2012)
20. Shtadler, Z., Grumberg, O.: Network Grammars, Communication Behaviors and Automatic Verification. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 151–165. Springer, Heidelberg (1990)
21. Trefler, R.J., Wahl, T.: Extending Symmetry Reduction by Exploiting System Architecture. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 320–334. Springer, Heidelberg (2009)

State Space Reduction for Sensor Networks Using Two-Level Partial Order Reduction*

Manchun Zheng[1], David Sanán[2], Jun Sun[1], Yang Liu[3], Jin Song Dong[4], and Yu Gu[1]

[1] Singapore University of Technology and Design
{zmanchun,sunjun,jasongu}@sutd.edu.sg
[2] School of Computer and Statistics, Trinity College Dublin
David.Sanan@cs.tcd.ie
[3] School of Computer Engineering, Nanyang Technological University
yangliu@ntu.edu.sg
[4] School of Computing, National University of Singapore
dongjs@comp.nus.edu.sg

Abstract. Wireless sensor networks may be used to conduct critical tasks like fire detection or surveillance monitoring. It is thus important to guarantee the correctness of such systems by systematically analyzing their behaviors. Formal verification of wireless sensor networks is an extremely challenging task as the state space of sensor networks is huge, e.g., due to interleaving of sensors and intra-sensor interrupts. In this work, we develop a method to reduce the state space significantly so that state space exploration methods can be applied to a much smaller state space without missing a counterexample. Our method explores the nature of networked NesC programs and uses a novel two-level partial order reduction approach to reduce interleaving among sensors and intra-sensor interrupts. We define systematic rules for identifying dependence at sensor and network levels so that partial order reduction can be applied effectively. We have proved the soundness of the proposed reduction technique, and present experimental results to demonstrate the effectiveness of our approach.

1 Introduction

Sensor networks (SNs) are built based on small sensing devices (i.e., sensors) and deployed in outdoor or indoor environments to conduct different tasks. Recently, SNs have been applied in more areas like military surveillance, environment monitoring, theft detection, and so on [2]. Many of them are carrying out critical tasks, failures or errors of which might cause catastrophic loss of money, equipments and even human lives. Therefore, it is highly desirable that the implementation of SN systems is reliable and correct.

In order to develop reliable and correct SNs, a variety of approaches and tools have been proposed. Static analysis of SNs (e.g., [3]) is difficult, given their dynamic nature. Therefore, most of the existing approaches rely on state space exploration, e.g., through simulation [15], random walk [17], or model checking [13,19,20,23,4,5,17]. Although

* This research is partially supported by project IDG31100105/IDD11100102 from Singapore University of Technology and Design.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 515–535, 2013.

some of the tools were able to detect and reveal bugs, all of them face the same challenge: the huge state space of SNs. In practice, a typical sensor program might consist of hundreds/thousands of lines of code (LOC), which introduces a state space of tens of thousands, considering only concurrency among internal interrupts. As a result, existing tools usually cover only a fraction of the state space and/or take a long time. For instance, the work in [4,5] is limited to a single sensor, whereas the approaches in [13,19,17,25] work only for small networks. We refer the readers to Section 6 for a detailed discussion of related works.

In this work, we develop a method to significantly reduce the state space of SNs while preserving important properties so that state space exploration methods (like model checking or systematic testing) become more efficient. Our targets are SNs developed in TinyOS/NesC, since TinyOS/NesC is widely used for developing SNs. TinyOS [7] provides an interrupt-driven execution model for SNs, and NesC (Network-Embedded-System C) [10] is the programming language of TinyOS applications.

Our method is a novel two-level partial order reduction (POR) which takes advantage of the unique features of SNs as well as NesC/TinyOS. Existing POR methods [11,6,9,24,12] reduce the state space of concurrent systems by avoiding unnecessary interleaving of *independent* actions. In SNs, there are two sources of "concurrency". One is the interleaving of different sensors, which would benefit from traditional POR. The other is introduced by the internal interrupts of sensors. An interrupt can occur anytime and multiple interrupts may occur in any sequence, producing numerous states. Applying POR for interrupts is highly nontrivial because all interrupts would modify the task queue and lead to different orders of scheduled tasks at run time. Our method extends and combines two different POR methods (one for intra-sensor interrupts and one for inter-sensor interleaving) in order to achieve better reduction. We remark that applying two different POR methods in this setting is complicated, due to the interplay between inter-sensor message passing and interrupts within a sensor (e.g., a message arrival would generate interrupts).

Our method preserves both safety properties and liveness properties in the form of linear temporal logic (LTL) so that state space exploration methods can be applied to a much-smaller state space without missing a counterexample. Our method works as follows. First, static analysis is performed to automatically identify independent actions/interrupts at both inter-sensor and intra-sensor levels. Second, we extend the Cartesian semantics [12] to reduce network-level interleaving. The original Cartesian POR algorithm treats each process (in our case, sensor) as a simple sequential program. However, in our work, we have to handle the internal concurrency among interrupts for each sensor and thus the Cartesian semantics of SNs is developed. The interleaving among interrupts is then minimized by the persistent set technique [6].

We formally prove that our method is sound and complete, i.e., preserving LTL-X properties [6]. The proposed method has been implemented in the model checker NesC@PAT [25] and experiment results show that our method reduces the state space significantly, e.g., the reduced state space is orders of magnitudes smaller. We also approximated the reduction ratio obtained by a related tool T-Check [17] under POR setting and the data show that our two-level POR achieves much better reduction ratio than T-Check's POR algorithm, as elaborated in Section 5.

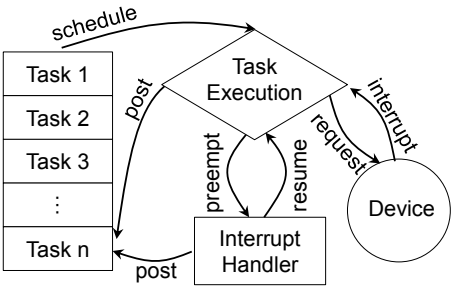
2 Preliminaries

In this section, we present the interrupt-driven feature of TinyOS/NesC and the formal definitions of SNs. For details on how to generate a model from NesC programs, readers are referred to [25], which defines small step operational semantics for NesC.

In NesC programs, there are two execution contexts, *interrupt handler* and *task* (a function), described as *asynchronous* and *synchronous*, respectively [10]. An interrupt handler can always preempt a task if interrupts are enabled. In TinyOS execution model [14], a task queue schedules tasks for execution in a certain order. In our work, we model the task scheduler in the FIFO order, which is the default setting of TinyOS and is widely used. As shown in Fig. 1(a), the execution of a task could be preempted by interrupt handlers. An interrupt handler accesses low-level registers and enqueues a task to invoke a certain function at a higher level of application code. In our approach, we treat interrupt handlers as black boxes, as we assume that the low-level behaviors of devices work properly. Variables are used to represent the status of a certain device and thus low-level functions related to interrupt handlers are abstracted, as shown by the pseudo code in Fig. 1(b). The execution of an interrupt handler is modeled as one action. Different ordering of interrupt handler executions might lead to different orders of tasks in the task queue, making the state space complex and large. In our model after a task is completed, all pending interrupt handlers are executed before a new task is loaded for execution. This approximation reduces concurrency between tasks and interrupts and is yet reasonable since devices usually respond to requests within a small amount of time like the executing period of a task.

The NesC language is an event-oriented extension of C that adds new concepts such as $call$, $signal$, and $post$. The semantics of $call$ (e.g., lines 2 and 10 in Fig. 1(c)) and that of $signal$ are similar to traditional function calls, invoking certain functions (either commands or events). The keyword $post$ (like lines 3 and 17 in Fig. 1(c)) is to enqueue a given task. Thus the task queue could be modified during both synchronous and asynchronous execution contexts. In other words, the task queue is shared by tasks and interrupt handlers. Fig. 1(c) illustrates a fragment of a NesC program, which involves messaging and sensing and is the running example of this paper. The command call $Read.read()$/$Send.send()$ invokes the corresponding command body that requests the sensing device/messaging device to read data/to send a packet, which will later trigger the completion interrupt rd/sd to post a task for signaling event $Read.rdDone$/$Send.sendDone$. We remark that rv is used to denote the interrupt of a packet arrival, and t_{rd}, t_{sd}, and t_{rv} are the tasks posted by interrupt handlers of rd, sd and rv, respectively. With the assumption that a packet arrival interrupt is possible at any time, the state graph of event $Boot.booted$ is shown in Fig. 1(d), where each transition is labeled with the line number of the executed statement or the triggered interrupt, and each state is numbered according to the task queue. The task queues of different state numbers are illustrated in Fig. 1(d) as well. For example, after executing $call\ Read.read()$ (line 2) the task queue still remains empty, while after executing the interrupt handler rv which enqueues its completion task t_{rv} and the task queue becomes $\langle t_{rv} \rangle$ (i.e., state 6).

The formal definitions of SNs are given in [25]. They are summarized below only to make the presentation self-contained.



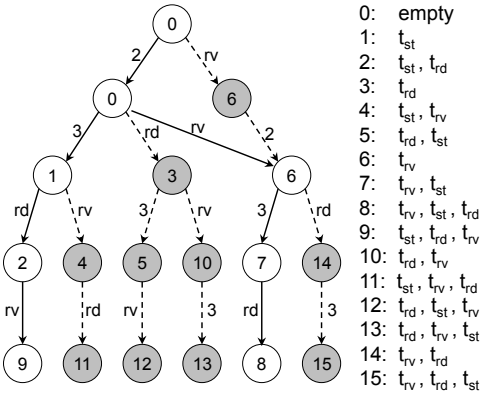
(a) TinyOS Execution Model

```
1  void handler_dev(){
2
3    //update status of dev;
4
5    ...
6
7    //schedule the
8
9    //completion event
10
11   post dev_compl_task();
12
13 }
14
15
16
17 //completion task
18
19 task void dev_compl_task(){
20
21   //event implemented
22
23   //by programmers
24
25   signal dev_done_event();
26
27 }
```

(b) Abstract Interrupt Handler

```
1  event void Boot.booted(){
2
3    call Read.read();
4
5    post send_task();
6
7  }
8
9  event void Read.rdDone(int v){
10
11   value += v;
12
13 }
14
15 task void send_task(){
16
17   busy = true;
18
19   call Send.send(count);
20
21 }
22
23 event void Send.sendDone(){
24
25   busy = false;
26
27 }
28
29 event void Receive.receive(){
30
31   count ++;
32
33   post send_task();
34
35 }
```

(c) Example Code



(d) State Graph of Event *Boot.booted*

Fig. 1. Interrupt-driven Features

Definition 1 (Sensor Model). *A sensor model $\mathcal{S}$ is a tuple $\mathcal{S} = (A, T, R, init, P)$ where A is a finite set of variables; T is a queue which stores posted tasks in FIFO order; R is a buffer that keeps incoming messages sent by other sensors; $init$ is the initial state of $\mathcal{S}$; and P is a program composed by the running NesC program M and interrupting devices H, i.e., $P = M \triangle H$.*

Definition 1 formally describes a sensor which runs NesC programs. Let $\mathcal{S}$ be a sensor. A state C of $\mathcal{S}$ is a tuple (V, Q, B, P) where V is the current valuation of variables declared by the NesC programs of $\mathcal{S}$; Q is a sequence of tasks scheduled in the task queue; B is the sequence of packets in the message buffer; and P is the running program. In this work, we use $V(C)$, $Q(C)$, $B(C)$ and $P(C)$ to denote the variable valuation, task queue, message buffer and running program of a state C, respectively.

A sensor transition t is defined as $C \xrightarrow{\alpha}_s C'$, where C (C') is the state before and after executing the action α, represented as $C' = ex(C, \alpha)$. We define $enable(C)$ to be the set of all actions enabled at state C, i.e., $enable(C) = \{\alpha \mid \exists\, C' \in \mathbb{C}, C \xrightarrow{\alpha} C'\}$. Further, $ex(C, \alpha)$ (where $\alpha \in enable(C)$) denotes the state after executing α at state C. $\sum_{\mathcal{S}}$ (or simply $\sum$ if $\mathcal{S}$ is clear) denotes the set of actions of $\mathcal{S}$. We define $itrQ(\mathcal{S}) \subseteq \sum$ as the set of hardware request actions and $sd(\mathcal{S})$ as the set of actions involving packet transmission. $Tasks(\mathcal{S})$ (or simply $Tasks$ if $\mathcal{S}$ is clear) denotes the set of all tasks defined in $\mathcal{S}$. For a given NesC program, we assume that $\sum$ and $Tasks$ are finite.

Definition 2 (Sensor Network Model). *A sensor network model $\mathcal{N}$ is defined as a tuple $(\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ where $\mathcal{R}$ is the network topology, and $\{\mathcal{S}_0, \cdots, \mathcal{S}_n\}$ is a finite ordered set of sensor models, with $\mathcal{S}_i$ ($0 \leqslant i \leqslant n$) being the i^{th} sensor model.*

A state $\mathcal{C}$ of a sensor network is defined as an ordered set of states $\{C_1, \cdots, C_n\}$ where C_i ($1 \leqslant i \leqslant n$) is the state of $\mathcal{S}_i$, denoted by $\mathcal{C}[i]$. A sensor network transition $\mathcal{T}$ is defined as $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ where $\mathcal{C}$ ($\mathcal{C}'$) is the state before (after) the transition, represented as $\mathcal{C}' = Ex(\mathcal{C}, \alpha)$. In the following of this paper, a state of a sensor is referred to as a *local* state, whereas a state of a sensor network is called a *global* state or simply a state.

3 Two-Level Independence Analysis

Inside a sensor, the interleaving between an interrupt handler and a non-post action can be reduced, since interrupt handlers only modify the task queue and non-post actions never access the task queue. For example, in Fig. 1(d), the interleaving between line 2 and rv can be ignored. Moreover, for post statements and interrupt handlers, their interleaving could be reduced if their corresponding tasks access no common variables. For example, t_{rd} only accesses variable $value$ which is never accessed by t_{rv}, so the interleaving between interrupt handlers rd and rv at state 1 can be alleviated. In Fig. 1(d), dashed arrows and shadow states stand for transitions and states that can be pruned. Therefore, it is important to detect the independence among actions inside a sensor, referred to as *local independence*.

Among a sensor network, each sensor only accesses its own and local resources, unless it sends a message packet, modifying some other sensors' message buffers. Intuitively, the interleaving of local actions of different sensors can be reduced without

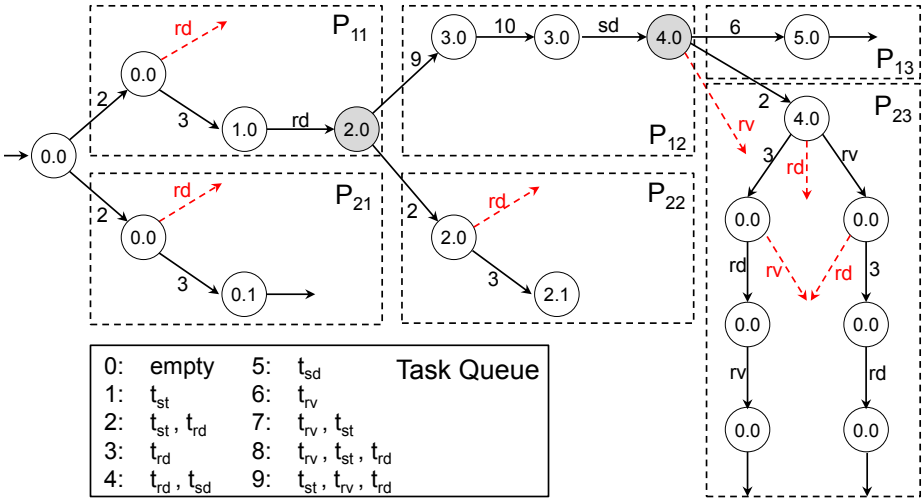


Fig. 2. Pruned State Graph

missing critical states. This observation leads to the independence analysis at the network level, referred to as *global independence*.

Consider a network with two sensors $\mathcal{S}_1$ and $\mathcal{S}_2$ implemented with the code shown in Fig. 1(c). Applying partial order reduction at both network and sensor levels, we are able to obtain a reduced state graph as shown in Fig. 2. States are numbered with the task queues of both sensors. For example, state 2.1 shows that the task queue of $\mathcal{S}_1$ is $\langle t_{st}, t_{rd} \rangle$ and $\langle t_{st} \rangle$ for $\mathcal{S}_2$. In this example, interleaving between the two sensors is only allowed when necessary, like at the shadow states labeled with 2.0 and 4.0. The sub-graph within each dashed rectangle is established by executing actions from only one sensor, either $\mathcal{S}_1$ or $\mathcal{S}_2$. In each sub-graph, local independence is applied to avoid unnecessary interleaving among local actions. Dashed arrows indicate pruned local actions. For example, rectangle P_{23} is constructed by removing all shadow states and dashed transitions in Fig. 1(d). The corresponding complete state space of this graph consists of around 200 states, whereas the reduced graph contains fewer than 20 states.

The shape of the reduced state graph might be different according to the property being checked. For example, if the property is affected by the value of the variables *busy* and *value* of the example in Fig. 1(d), then the interleaving between rd and rv can not be avoided. Therefore, we need to investigate local and global independence w.r.t. a certain property, and in the following of this paper, the concepts of independence, equivalence and so on are discussed w.r.t. a certain property φ. We present the definitions of local independence and global independence in Section 3.1 and 3.2, respectively, both with rules for identifying them.

3.1 Local Independence

In a sensor, an action may modify a variable or the task queue. Local independence is defined by the effects on the variables and the task queue. In the following, the concepts of actions, tasks, and task queues are w.r.t. a given sensor $\mathcal{S}$.

Definition 3 (Local Independence). *Given a local state C, $\alpha_1, \alpha_2 \in \sum$, and $\alpha_1, \alpha_2 \in enable(C)$. Actions α_1 and α_2 are said to be local-independent, denoted by $\alpha_1 \equiv_{LI} \alpha_2$, if the following conditions are satisfied:*

1. $ex(ex(C, \alpha_1), \alpha_2) =_v ex(ex(C, \alpha_2), \alpha_1)$;
2. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$.

In the above definition, $=_v$ denotes that two local states share the same valuation of variables, message buffer, and the same running program. That is, if $C_1 = (V_1, Q_1, B_1, P_1)$, $C_2 = (V_2, Q_2, B_2, P_2)$, then we have $C_1 =_v C_2$ iff $V_1 = V_2 \wedge B_1 = B_2 \wedge P_1 = P_2$. If only the first condition in Definition 3 is satisfied, α_1 and α_2 are said to be *variable-independent*, denoted by $\alpha_1 \equiv_{VI} \alpha_2$. The relation $\simeq$ will be covered in Definition 6. Let W_α and R_α be the set of variables written and only read by an action α, respectively.

Lemma 1. $\forall \alpha_1, \alpha_2 \in \sum . \ W_{\alpha_1} \cap (W_{\alpha_2} \cup R_{\alpha_2}) = W_{\alpha_2} \cap (W_{\alpha_1} \cup R_{\alpha_1}) = \varnothing \Rightarrow \alpha_1 \equiv_{VI} \alpha_2$. □

Lemma 1 [1] shows that two actions are variable-independent if the variables modified by one action are mutually exclusive with those accessed (either modified or read) by the other. For example, $\alpha_{l6} \equiv_{VI} \alpha_{l13}$, where $\alpha_{l6}(\alpha_{l13})$ refers to the action executing the statement at line 6 (13) of Fig. 1(c). Furthermore, we can conclude that a non-post action in the synchronous context is always local-independent with any action in the asynchronous context [1]. This is shown in Lemma 2.

Lemma 2. $\forall \alpha \in \sum^{syn}, \alpha' \in \sum^{asyn} . \ \alpha \notin \sum^{pt} \Rightarrow \alpha \equiv_{LI} \alpha'$. □

Inside a sensor, interrupt handlers might run in parallel, which produces different orders of tasks in the task queue. Given a task t, $Ptask(t)$ denotes the set of tasks posted by a post statement in t or an interrupt handler of a certain interrupt request in t. Formally, $Ptask(t) = \{t' \mid \exists \alpha \in t. \ \alpha = post(t') \vee (\alpha \in itrQ(\mathcal{S}) \wedge t' = tsk(ih(\alpha)))\}$, where $post(t)$ is a post statement to enqueue task t; $ih(\alpha_{iq})$ denotes the corresponding interrupt handler of a device request α_{iq}, and $tsk(\alpha_{ih})$ denotes the completion task of α_{ih}. In the code in Fig. 1(c), $Ptask(t_{rv}) = \{t_{st}\}$, due to the $post$ statement in line 17. As for t_{st} (lines 8 to 11), it has a request for sending a message (line 10), the interrupt handler of which will post the task t_{sd}, and thus $Ptask(t_{st}) = \{t_{sd}\}$.

Since more tasks can be enqueued during the execution of a previously enqueued task, we define $Rtask(t)$ to represent all possible tasks enqueued by a given task t and the tasks in its $Ptask$ set in a recursive way. Formally, $Rtask(t) = \{t\} \cup Ptask(t) \cup (\cup_{t' \in Ptask(t)} Rtask(t'))$. Since $Tasks$ is finite, for every task t, $Rtask(t)$ is also finite and thus could be obtained statically at compile time. In Fig. 1(c), since $Ptask(t_{sd}) = \varnothing$, we have $Rtask(t_{sd}) = \{t_{sd}\}$. Similarly, we can obtain that $Rtask(t_{st}) = \{t_{st}, t_{sd}\}$ and $Rtask(t_{rv}) = \{t_{rv}, t_{st}, t_{sd}\}$. Let $R(\varphi, \mathcal{S})$ be the set of variables of $\mathcal{S}$ accessed by the property φ. Let $\widehat{W}(t)$ be the set of variables modified by any task in $Rtask(t)$. We say that t is a φ-safe task, denoted by $t \in safe(\varphi, \mathcal{S})$ iff $(\widehat{W}(t) \cap R(\varphi, \mathcal{S})) = \varnothing$.

Definition 4 (Local Task Independence). *Let $t_{i(j)} \in Tasks$ be two tasks. t_i and t_j are said to be local-independent, denoted by $t_i \equiv_{TI} t_j$, iff $(t_i \in safe(\varphi, \mathcal{S}) \vee t_j \in safe(\varphi, \mathcal{S})) \wedge \forall t'_i \in Rtask(t_i), t'_j \in Rtask(t_j). \ \forall \alpha_i \in t'_i, \alpha_j \in t'_j. \ \alpha_i \equiv_{VI} \alpha_j$.*

Though interrupt handlers and post statements might modify the task queue concurrently, we observe that task queues with different orders of tasks might be equivalent. Based on Definition 4, we define the independence relation of two task sequences, which is used to further define equivalent task sequences.

Definition 5 (Task Sequence Independence). *Let* $Q_i = \langle t_{i0}, \cdots, t_{im}\rangle$, $Q_j = \langle t_{j0}, \cdots, t_{jn}\rangle (m, n \geqslant 0)$ *be two task sequences, where* $t_{iu} (0 \leq u \leq m), t_{jv} (0 \leq v \leq n) \in Tasks$. Q_i *and* Q_j *are said to be sequence-independent, denoted by* $Q_i \equiv_{SI} Q_j$, *iff* $\forall\, t_i \in (\cup_{k=0}^{m} Rtask(t_{ik})), t_j \in (\cup_{k=0}^{n} Rtask(t_{jk}))$. $t_i \equiv_{TI} t_j$.

Let $q_1 \frown q_2$ be the concatenation of two sequences q_1 and q_2. A partition $\mathcal{P}$ of a task sequence Q is a list of task sequences $q_0, q_1, \cdots, q_m$ such that $Q = q_0 \frown q_1 \frown \cdots q_m$, and for all $0 \leqslant i \leqslant m$, $q_i \neq \langle\rangle$ (q_i is called a sub-sequence of Q). We use $part(Q)$ to denote the set of all possible partitions of Q. Given a partition $\mathcal{P}$ of Q such that $Q = q_0 \frown q_1 \frown \cdots \frown q_n$, $Swap(Q, i) = q_0 \frown \cdots q_{i+1} \frown q_i \frown \cdots \frown q_n$ denotes the task sequence obtained by swapping two adjacent sub-sequences (i.e., q_i and q_{i+1}) of Q.

Definition 6 (Task Sequence Equivalence). *Two task sequences* Q *and* Q' *are equivalent (*$Q \simeq Q'$*) iff* $Q^0 = Q \wedge \exists\, m \geqslant 0$. $Q^m = Q' \wedge (\forall\, 0 \leq k < m.\ (\exists\, i_k.\ Q^{k+1} = Swap(Q^k, i_k) \wedge q^k_{i_k} \equiv_{SI} q^k_{i_k+1}))$ *where* q^k_i *is the* i^{th} *sub-sequence of* Q^k.

The above definition indicates that if a task sequence Q' can be obtained by swapping adjacent independent sub-sequences of Q, then $Q \simeq Q'$. Given two local states C and C', we said that C is equivalent to C', denoted by $C \cong C'$, iff $C =_v C' \wedge Q(C) \simeq Q(C')$. Further, two local state sets $\mathbb{C}, \mathbb{C}'$ are said to be equivalent, denoted by $\mathbb{C} \asymp \mathbb{C}'$, iff $\forall\, C \in \mathbb{C}.\ \exists\, C' \in \mathbb{C}'.\ C \cong C'$ and vice versa. We explore the execution of task sequences starting at a local state which is the completion point of a previous task, i.e., a local state with the program as $(\checkmark \triangle H)$ [25]. This is because that only after a task terminates can a new task be loaded from the task queue for execution. The case when $B(C) \neq \langle\rangle$ is related to network communication, and is ignored here but will be covered in global independence analysis in Section 3.2.

Lemma 3. *Given* $C = (V, Q, \langle\rangle, \checkmark \triangle H)$ *and* $C' = (V, Q', \langle\rangle, \checkmark \triangle H)$, *let* $exs(Q_i, C_i)$ *be the set of final local states after executing all tasks of* Q_i *starting at local state* C_i. $Q \simeq Q' \Rightarrow exs(Q, C) \asymp exs(Q', C')$. □

Lemma 3 shows that executing two equivalent task sequences from v-*equal* local states will always lead to equivalent sets of final local states, as proved in [1]. Given an action α, we use $ptsk(\alpha)$ to denote the set of tasks that could be enqueued by executing α. With the above lemma, the rule for deciding local independence between actions can be obtained by Lemma 4 [1].

Lemma 4. *Given* C, $\alpha_1, \alpha_2 \in enable(C)$, $(\alpha_1 \equiv_{VI} \alpha_2 \wedge \forall\, t_1 \in ptsk(\alpha_1), t_2 \in ptsk(\alpha_2).\ t_1 \equiv_{TI} t_2) \Rightarrow \alpha_1 \equiv_{LI} \alpha_2$. □

3.2 Global Independence

SNs are non-blocking, i.e., the execution of one sensor never blocks others. In addition, a sensor accesses local resources most of the time, except when it broadcasts a message

to the network and fills in others' message buffers. At the network level, we explore the execution of each sensor individually, and only allow interleaving among sensors when an action involving network communication is performed. Let $\mathcal{N}$ be a sensor network with n sensors $\mathcal{S}_1, \mathcal{S}_2 \cdots \mathcal{S}_n$ and $\mathcal{C}$ be a global state. We use $EnableT(\mathcal{C})$ to denote the set of enabled tasks at $\mathcal{C}$. Given $t \in Tasks(\mathcal{S}_i)$, $t \in EnableT(\mathcal{C}) \Leftrightarrow \mathcal{C}[i] = (V, \langle t, \cdots \rangle, B, \checkmark \triangle H)$. $Ex(\mathcal{C}, t)$ represents the set of final states after executing task t (and interrupt handlers caused by it) starting from $\mathcal{C}$. For two global states $\mathcal{C}_1$ and $\mathcal{C}_2$, we say that $\mathcal{C}_1$ and $\mathcal{C}_2$ are equivalent ($\mathcal{C}_1 \cong \mathcal{C}_2$) iff $\forall\, 1 \leq i \leq n.\ \mathcal{C}_1[i] \cong \mathcal{C}_2[i]$. Similarly, we say that two sets of global states Γ and Γ' are equivalent (i.e., $\Gamma \asymp \Gamma'$) iff $\forall\, \mathcal{C} \in \Gamma.\ \exists\, \mathcal{C}' \in \Gamma'.\ \mathcal{C} \cong \mathcal{C}'$ and vice versa.

Definition 7 (Global Independence). *Let $t_i \in Tasks(\mathcal{S}_i)$ and $t_j \in Tasks(\mathcal{S}_j)$ such that $\mathcal{S}_i \neq \mathcal{S}_j$. Tasks t_i and t_j are said to be global-independent, denoted by $t_i \equiv_{GI} t_j$, iff $\forall\, \mathcal{C} \in \Gamma.\ t_i, t_j \in EnableT(\mathcal{C}) \Rightarrow \forall\, \mathcal{C}_i \in Ex(\mathcal{C}, t_i).\ \exists\, \mathcal{C}_j \in Ex(\mathcal{C}, t_j).\ Ex(\mathcal{C}_i, t_j) \asymp Ex(\mathcal{C}_j, t_i)$ and vice versa.*

A data transmission would trigger a packet arrival interrupt at the receivers and thus is possible to interact with local concurrency inside sensors. In the following, $Sends(\mathcal{S})$ denotes the set of tasks that contain data transmission requests, and $Rcvs(\mathcal{S})$ denotes the set of completion tasks of packet arrival interrupts.

Given $t \in Tasks(\mathcal{S})$, t is considered as rcv-independent, denoted by $t \subset_{RI} \mathcal{S}$, iff $\forall\, t_r \in Rcvs(\mathcal{S}), t_p \in Posts(t).\ t_r \equiv_{TI} t_p$. A rcv-independent task never posts a task local-dependent with the completion task of any packet arrival interrupts. Thus, we can ignore interleaving such tasks with other sensors even if there exists data transmission. We say that t is a *global-safe* task of $\mathcal{S}$, i.e., $t \subset_{GI} \mathcal{S}$, iff $t \subset_{RI} \mathcal{S}$. If $t \not\subset_{GI} \mathcal{S}$, then t is *global-unsafe*. The following theorem indicates that a global-safe task is always global-independent with any task of other sensors [1].

Theorem 1. $\forall\, t_1 \in Tasks(\mathcal{S}_i), t_2 \in Tasks(\mathcal{S}_j).\ \mathcal{S}_i \neq \mathcal{S}_j, t_1 \subset_{GI} \mathcal{S}_i \Rightarrow t_1 \equiv_{GI} t_2.$ □

4 SN Cartesian Partial Order Reduction

In this section, we present our two-level POR, which extends the Cartesian vector approach [12] and combines it with a persistent set algorithm [11].

4.1 Sensor Network Cartesian Semantics

Cartesian POR was proposed by Gueta et. al. to reduce non-determinism in concurrent systems, which delays unnecessary context switches among processes [12]. Given a concurrent system with n processes and a state s, a Cartesian vector is composed by n prefixes, where the i^{th} ($1 \leq i \leq n$) prefix refers to a trace executing actions only from the i^{th} process starting from state s. For SNs, sensors could be considered as concurrent processes and their message buffers could be considered as "global variables".

It has been shown that Cartesian semantics is sound for local safety properties [12]. A global property that involves local variables of multiple processes (or sensors) is converted into a local property by introducing a dummy process for observing involved

variables. In our case, we avoid this construction by considering global property in the Cartesian semantics for SNs. Let $Gprop(\mathcal{N})$, or simply $Gprop$ since $\mathcal{N}$ is clear in this section, be the set of *global* properties defined for $\mathcal{N}$. Given an action $\alpha \in Tasks(\mathcal{S})$ and a global property $\varphi \in Gprop$, α is said to be *φ-safe*, denoted by $\alpha \in safe(\varphi)$, iff $W_\alpha \cap R(\varphi) = \varnothing$ where W_α is the set of variables modified by α and $R(\varphi)$ is the set of variables accessed by φ. If $\alpha \notin safe(\varphi)$, then α is said to be *φ-unsafe*.

In order to allow sensor-level nondeterminism inside prefixes, we redefine $Prefix$ as a "trace tree" rather than a sequential trace. Let $Prefix(\mathcal{S})$ be the set of all prefixes of sensor $\mathcal{S}$, $Prefix(\mathcal{S}, \mathcal{C})$ be the set of prefixes of $\mathcal{S}$ starting at $\mathcal{C}$, and $first(p)$ be the initial state of a prefix p. A prefix is defined as follows.

Definition 8 (Prefix). *A prefix $p \in Prefix(\mathcal{S})$ is defined as a tuple $(trunk, branch)$, where $trunk = \langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m \rangle \wedge m \geq 0 \wedge \forall 1 \leq i < m.\ \alpha_i \in \sum_{\mathcal{S}} \wedge \mathcal{C}_i \xrightarrow{\alpha_i} \mathcal{C}_{i+1}$, and $branch \subseteq Prefix(\mathcal{S}, \mathcal{C}_m)$, being a set of prefixes of $\mathcal{S}$.*

Let $p \in Prefix(\mathcal{S})$ and $p = (p_{tr}, \{p_{b1}, p_{b2}, \cdots, p_{bm}\})$. We define $tr(p)$ to denote the trunk of prefix p before branching prefixes (i.e., $tr(p) = p_{tr}$), and $br(p)$ to denote the set of branching prefixes of p (i.e., $br(p) = \{p_{b1}, p_{b2}, \cdots, p_{bm}\}$). In Fig. 2, the dashed rectangles p_{11}, p_{12} and p_{13} are prefixes of $\mathcal{S}_1$, and p_{21}, p_{22} and p_{23} are prefixes of $\mathcal{S}_2$. More specifically, $tr(p_{23}) = \langle (4.0), \alpha_2, (4.0) \rangle$ and $br(p_{23}) = \{\langle (4.0), \alpha_3, (4.1), \alpha_{rd}, \cdots \rangle, \langle (4.0), \alpha_{rv}, (4.6), \alpha_3, \cdots \rangle\}$. Given a prefix $p \in Prefix(\mathcal{S})$, the following notations are defined:

- The set of states in p: $states(p) = \{\mathcal{C}_0, \cdots, \mathcal{C}_m\} \cup (\cup_{sp \in br(p)} states(sp))$.
- The set of *leaf* prefixes of $\mathcal{S}$: $LeafPrefix(\mathcal{S}) = \{lp \mid \forall\, lp \in Prefix(\mathcal{S}).\ br(lp) = \varnothing\}$. Given $lp \in LeafPrefix(\mathcal{S})$, $\widehat{last}(lp)$ denote the last state of lp.
- The set of *tree* prefixes of $\mathcal{S}$: $TreePrefix(\mathcal{S}) = Prefix(\mathcal{S}) - LeafPrefix(\mathcal{S})$.
- The set of leaf prefixes of p: $p \in LeafPrefix(\mathcal{S}) \Rightarrow leaf(p) = p \wedge p \in TreePrefix(\mathcal{S}) \Rightarrow leaf(p) = \cup_{bp \in br(p)} leaf(bp)$.
- The set of final states of p: $p \in LeafPrefix(\mathcal{S}) \Rightarrow last(p) = \{\widehat{last}(p)\} \wedge p \in TreePrefix(\mathcal{S}) \Rightarrow last(p) = \cup_{bp \in br(p)} last(bp)$.
- Subsequent prefixes $\sqsupset$: $\forall\, lp \in LeafPrefix(\mathcal{S}).\ lp \sqsupset p \equiv \widehat{last}(lp) = first(p)$.
- Concatenation of leaf prefixes $\frown$: $\forall\, p1 = \langle \mathcal{C}_0, \alpha_0, \cdots, \mathcal{C}_k \rangle, p2 = \langle \mathcal{C}_k, \alpha_k, \mathcal{C}_{k_0}, \alpha_{k_0}, \cdots, \mathcal{C}_{k_m} \rangle. p1 \frown p2 = \langle \mathcal{C}_0, \alpha_0, \cdots, \mathcal{C}_k, \alpha_k, \mathcal{C}_{k_0}, \cdots, \mathcal{C}_{k_m} \rangle$.

We also define $tasks(p)$ ($acts(p)$) to denote the set of tasks (actions) executed in p. Moreover, $lastT(p)$ ($lastAct(p)$) denotes the set of last tasks (actions) executed in p.

Definition 9 (SN Cartesian Vector). *Given a global property $\varphi \in Gprop$, a vector $(p_1, \cdots, p_i, \cdots, p_n) \in Prefix^n$ is a sensor network Cartesian vector for $\mathcal{N}$ w.r.t. φ from a state $\mathcal{C}$* if the following conditions hold*:*

1. $p_i \in Prefix(\mathcal{S}_i, \mathcal{C})$;
2. $\forall\, t \in tasks(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in LastT(p_i)$;
3. $\forall\, \alpha \in acts(p_i).\ \alpha \notin safe(\varphi) \Rightarrow \alpha \in lastAct(p_i)$.

According to Definition 9, a vector $(p_0, p_1, \cdots, p_n)$ from $\mathcal{C}$ is a valid sensor network Cartesian vector (SNCV) if for every $0 \leq i \leq n$, p_i is a prefix of $\mathcal{S}_i$ and each leaf prefix

Algorithm 1. State Space Generation

$GetSuccessors(\mathcal{C}, p, \varphi)$

1: $list \leftarrow \varnothing$
2: **if** $Next(p, \mathcal{C}) \neq \varnothing$ **then**
3: $\quad list \leftarrow Next(p, \mathcal{C})$
4: **else**
5: $\quad scv \leftarrow GetNewCV(\mathcal{C}, \varphi)$
6: $\quad$ **for all** $i \leftarrow 1$ **to** n **do**
7: $\quad\quad list \leftarrow list \cup \{Next(scv[i], \mathcal{C})\}$
8: $\quad$ **end for**
9: **end if**
10: **return** $list$

of p_i ends with a φ-unsafe action or a global-unsafe task as defined in Section 3.2. Furthermore, we define the corresponding inference rules of SNCVs [1]. In Fig. 2, if $value, busy \notin R(\varphi)$, then (p_{11}, p_{21}) is a valid SNCV from the initial state.

4.2 Two-Level POR Algorithm

In this section, we present the two-level POR algorithm. The main idea is to explore the state space by the sensor network Cartesian semantics and to perform reduction during the generation of SNCVs. First, we present the top-level state exploration algorithm, which could be invoked in existing verification algorithms directly without changing the verification engine. Second, we show the algorithms for SNCV generation, as well as algorithms for producing a sensor prefix.

- **State Space Generation**. Given a state $\mathcal{C}$, a prefix p ($\mathcal{C} \in states(p)$) and a global property φ, the state space of $\mathcal{N}$ is explored via a corresponding SNCV, as shown in Algorithm 1. In this algorithm, $GetNewCV(\mathcal{C}, \varphi)$ generates a new SNCV from $\mathcal{C}$, which will be explained later. The relation $Next : Prefix(\mathcal{S}) \times \Gamma \rightarrow \mathbb{P}(\Gamma)$ traverses a prefix to find a set of successors of $\mathcal{C}$. Formally, $Next(p, \mathcal{C}) = \{\mathcal{C}' \mid \exists\, \alpha \in acts(p), \mathcal{C} \overset{\alpha}{\hookrightarrow} \mathcal{C}'\}$. The function $ConcatTree$ extends a leaf prefix with another prefix as its branch, defined as $ConcatTree(lp \in LeafPrefix(\mathcal{S}), sp \in Prefix(\mathcal{S}))$. Formally, if $lp \sqsupset sp$, after executing $ConcatTree(lp, sp)$, we have $lp' = (lp, \{sp\})$. We remark that $ConcatTree(lp, sp)$ has a side effect in lp by updating it with the resultant prefix of the combination.
- **SNCV Generation**. Algorithm 2 is dedicated to SNCV generation, i.e., the method $GetNewCV$. In this algorithm, $visited$ is the set of final states of prefixes that have been generated, and $workingLeaf$ is the stack of leaf prefixes to be further extended. Concurrency at network level is minimized by lines 7 and 18, where the relation $Extensible : Prefix(\mathcal{S}) \times \{\mathcal{S}_1, \cdots, \mathcal{S}_n\} \times Gprop \rightarrow \{True, False\}$ is defined as $Extensible(p, \mathcal{S}, \varphi) \equiv \forall\, t \in lastT(p), \alpha \in lastAct(p).\ t \subset_{GI} \mathcal{S} \wedge \alpha \in safe(\varphi) \wedge \alpha \notin sd(\mathcal{S})$. In other words, a prefix is further extended (lines 15 to 21) only if it has not executed a global-unsafe task, a φ-unsafe action or a messaging action. The function $GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$ produces a prefix of $\mathcal{S}_i$ by executing actions and interrupt handlers of $\mathcal{S}_i$ in parallel. At first, p_i is initialized by $GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$, and is then extended by recursively concatenating each of its leaf prefixes with a new prefix obtained by $GetPrefix(\mathcal{S}_i, \mathcal{C}', \varphi)$, as shown by lines 12 to 22. If p_i is inextensible, it is assigned to the i^{th} element of the sensor Cartesian vector scv ($scv[i]$) by line 23.

Algorithm 2. Sensor Network Cartesian Vector Generation

GetNewCV$(\mathcal{C}, \varphi)$

1: $scv \leftarrow (\overbrace{\langle\rangle, \cdots, \langle\rangle}^{n})$
2: **for all** $\mathcal{S}_i \in \mathcal{N}$ **do**
3: $visited \leftarrow \{\mathcal{C}\}$
4: $workingLeaf \leftarrow \varnothing$
5: $p_i \leftarrow GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$
6: **for all** $lp \in leaf(p_i)$ **do**
7: **if** $Extensible(lp, \mathcal{S}_i, \varphi)$ **and** $\widehat{last}(lp) \notin visited$ **then**
8: $workingLeaf.Push(lp)$
9: $visited = visited \cup \widehat{last}(lp)$
10: **end if**
11: **end for**
12: **while** $workingLeaf \neq \varnothing$ **do**
13: $p_k \leftarrow workingLeaf.Pop()$
14: $visited \leftarrow visited \cup \{\widehat{last}(p_k)\}$
15: $p'_k \leftarrow GetPrefix(\mathcal{S}_i, \widehat{last}(p_k), \varphi)$
16: $ConcatTree(p_k, p'_k)$
17: **for all** $lp \in leaf(p'_k)$ **do**
18: **if** $Extensible(lp, \mathcal{S}_i, \varphi)$ **and** $\widehat{last}(lp) \notin visited$ **then**
19: $workingLeaf.Push(lp)$
20: **end if**
21: **end for**
22: **end while**
23: $scv[i] \leftarrow p_i$
24: **end for**
25: **return** scv

Algorithm 3. Prefix Generation

GetPrefix$(\mathcal{S}, \mathcal{C}, \varphi)$

1: $p \leftarrow \langle \mathcal{C} \rangle$
2: $t \leftarrow getCurrentTsk(\mathcal{C}, \mathcal{S})$
3: $ExecuteTask(t, p, \varphi, \{\mathcal{C}\}, \mathcal{S})$
4: **if** t is finished **then**
5: **for all** $p_i \in leaf(p)$ **do**
6: $\mathcal{C}' \leftarrow \widehat{last}(p_i)$
7: $irs \leftarrow GetItrs(\mathcal{C}', \mathcal{S})$
8: $p'_i \leftarrow RunItrs(\mathcal{C}', itrs)$
9: $ConcatTree(p_i, p'_i)$
10: **end for**
11: **end if**
12: **return** p

- **Sensor Prefix Generation**. Algorithm 3 shows how a sensor $\mathcal{S}$ establishes a prefix from $\mathcal{C}$ w.r.t. φ. Function $ExecuteTask(t, p, \varphi, \mathcal{C}s, \mathcal{S})$ extends the initial prefix p by executing actions in task t, until a φ-unsafe action or a loop is encountered. Interrupt handlers are delayed as long as the action being executed is a non-post statement, which is reasonable due to Lemma 1 and Lemma 4. A persistent set approach has been adopted in both $ExecuteTask$ and $RunItrs$ to constrain interleaving to happen only between local-dependent actions.
- **Task Execution.** In Algorithm 4, the following notations are used.
  - Set $\mathcal{C}s$: the set of states that has been visited.
  - Method $GetAction(t, \mathcal{C})$: returns the enabled action of task t at state $\mathcal{C}$.
  - Method $setPfx(\mathcal{C}, p)$: assigns prefix p as the prefix that state $\mathcal{C}$ belongs to.

  Initially, the currently enabled action α will be executed (lines 5 to 16). At this phase, two cases are considered. The first is when α is a *post* statement, and interrupts dependent with α will be taken to run in parallel in order to preserve states with different task queues. This is achieved by lines 5 to 9. The second case handles all non-*post* actions, and the action will be executed immediately to obtain the resultant prefix (lines 10 to 16). In this case, all interleaving between interrupts and the action α is ignored, which is reasonable by Lemma 2. After the action α completes its execution, the algorithm will return immediately if α is φ-unsafe or t has no more actions to be executed. Otherwise, a new iteration of $ExecuteTask$

Algorithm 4. Task Execution

```
ExecuteTask(t, lp, φ, Cs, S)
 1: {let α be the current action of t}
 2: α ← GetAction(t, C)
 3: C ∈ last(lp)
 4: {only post actions need to
    interleave interrupts}
 5: if α ← post(t') then
 6:    itrs ← GetItrs(S, C)
 7:    {interleave α and interrupts itrs}
 8:    p ← RunItrs(C, itrs ∪ {α}, S)
 9:    lp ← (lp, {p})
10: else
11:    {non-post actions run independently}
12:    C' ← ex(C, α)
13:    tmp ← ⟨C, α, C'⟩
14:    setPfx(C', tmp)
15:    lp ← (lp, {tmp})
16: end if
17: lps ← leaf(lp)
18: {stop executing t when t terminates or
    a non-safe action is encountered}
19: if α ∉ safe(φ) or terminate(t, α) then
20:    return
21: end if
22: for all lp' ∈ lps do
23:    {extend lp only if there is no loop in it}
24:    if last(lp') ∉ Cs then
25:       Cs' ← Cs ∪ states(lp')
26:       {continue to execute t to extend lp'}
27:       ExecuteTask(t, lp', φ, Cs', S)
28:    end if
29: end for
```

will be invoked at each final state of the prefix that has been currently established (lines 22 to 29). Line 24 is to prevent the algorithm to be stuck by loops.

- **Interleaving Interrupts.** Algorithms 5 and 6 show how partial order reduction could be applied at sensor level to alleviate interleaving caused by concurrency among tasks and interrupts. The idea is motivated by the observation that the only shared resource among interrupt handlers and normal actions is the task queue. By Lemma 2, there are two kinds of concurrency to be considered, i.e. the concurrency between a *post* statement, and the concurrency between any two interrupt handlers. Algorithm 5 (*RunItrs*) establishes a prefix for the sensor $\mathcal{S}$ from a state $\mathcal{C}$ by interleaving actions in the set *itrs* using a persistent set approach. Here, *itrs* would be a set of interrupt handlers plus at most one *post* action. Algorithm 6 (Persistent Set) establishes a persistent set from a given set of actions *itrs*. If *itrs* contains a *post* action, then this *post* action will be chosen as the first action of the persistent set to return; otherwise, an action will be chosen randomly to start generating the persistent set (lines 2 to 10). After that, the persistent set will be extended by iteratively adding actions from *itrs* that are dependent with at least one action in the persistent set.

4.3 Correctness

In the following, we show that the above POR algorithms work properly and are sound for model checking global properties and LTL-X properties. Lemmas 5 and 6 assure the correctness of the functions invoked in Algorithm 3, which are proved in [1].

Lemma 5. *Given a state* $\mathcal{C}$ *where* $\mathcal{C}[i] = (V, Q, B, \checkmark \triangle H)$, *RunItrs*$(\mathcal{C}$, *Get-Itrs*$(\mathcal{C}', \mathcal{S}_i))$ *terminates and returns a valid prefix of* $\mathcal{S}_i$. □

Algorithm 5. Interleaving Interrupts

```
RunItrs(C, itrs, S)
 1: if itrs ← ∅ then
 2:     return ⟨⟩
 3: end if
 4: p ← ⟨C⟩
 5: {pis is the persistent set of itrs}
 6: pis ← GetPerSet(itrs, C, S)
 7: {interleave dependent actions}
 8: for all α ∈ pis do
 9:     C' ← ex(C, α)
10:     lp ← ⟨C, α, C'⟩
11:     setPfx(C', lp)
12:     {only allow interleaving if α is not a post}
13:     if α ∈ ∑^iq then
14:         s ← RunItrs(C', pis − {α}, S)
15:         lp ← (lp, {s})
16:     end if
17:     {add lp as a new branch to p}
18:     p ← (tr(p), br(p) ∪ {lp})
19: end for
20: return p
```

Algorithm 6. Persistent Set

```
GetPerSet(itrs, C, S)
 1: {choose an α to start with}
 2: if ∃ α' ∈ itrs. α ∉ ∑^iq then
 3:     {there exists a post in itrs,
        then we should start from the post}
 4:     α ← α'
 5: else
 6:     if α' ∈ itrs then
 7:         {choose an α form itrs randomly}
 8:         α ← α'
 9:     end if
10: end if
11: pset ← {α}
12: work ← {α}
13: while work ≠ ∅ do
14:     α ← work.Pop()
15:     {find new dependent actions of α from itrs}
16:     αs ← DepActions(α, itrs − pset)
17:     pset ← pset ∪ αs
18:     work ← work ∪ αs
19: end while
20: return pset
```

Lemma 6. *Given $t \in Tasks(\mathcal{S})$, $t \in EnableT(\mathcal{C})$ and φ, $ExecuteTask(t, \langle\mathcal{C}\rangle, \varphi, \{\mathcal{C}\})$ extends $\langle\mathcal{C}\rangle$ by executing actions in t and enabled interrupt handlers, until t terminates or a φ-unsafe action or a loop is encountered.* □

Based on Lemma 5 and 6, we can show the correctness of Algorithm 2 in generating a prefix for a given state and a property, as shown in the following theorem.

Theorem 2. *Given $\mathcal{S}$, $\mathcal{C}$ and φ, Algorithm 3 terminates and returns a valid prefix of $\mathcal{S}$ for some SNCV.*

Proof By Lemma 6, after line 3 p is a valid prefix of $\mathcal{S}$. If lines 5 to 10 are not executed, then p is immediately returned. Suppose lines 5 to 10 are executed, and at the beginning of the i^{th} iteration of the "for" loop p is a valid prefix. Let $\widehat{p}$ be the updated prefix after line 9, and then $leaf(\widehat{p}) = (leaf(p) - p_i) \cup leaf(p'_i)$ since p_i has been concatenated with p'_i. By Lemma 5, p'_i is a valid prefix and thus $\widehat{p}$ is a valid prefix. Therefore, at the beginning of the $(i+1)^{th}$ iteration, p is a valid prefix. By Lemmas 6 and 5, both lines 3 and 8 terminate. Further, we assume that variables are finite-domain, and thus the size of $leaf(p)$ is finite assuring that the "for" loop terminates. □

Theorem 3. *For every state $\mathcal{C}$, Algorithm 2 terminates and returns a valid sensor network Cartesian vector.*

Proof. We prove that at the beginning of each iteration of the "while" loop (lines 12 to 22) in Algorithm 2 the following conditions hold for any i ($1 \leq i \leq n$):

1. $p_i \in \mathit{Prefix}(\mathcal{S}_i, \mathcal{C})$;
2. $\mathit{workingLeaf} = \{p \in \mathit{leaf}(p_i) \mid \mathit{Extensible}(p, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(p) \notin \mathit{visited}\}$.

By line 5, it is immediately true that $\mathit{first}(p_i) = \mathcal{C}$. Since $\mathit{ConcatTree}$ never changes the first state of a given prefix, $\mathit{first}(p_i) = \mathcal{C}$ holds for all iterations. Since p_i is extended by $\mathit{GetPrefix}(\mathcal{S}_i, \widehat{\mathit{last}}(p_k), \varphi)$ (line 15), which only executes actions of $\mathcal{S}_i$, thus $p_i \in \mathit{Prefix}(\mathcal{S}_i)$ always holds. Intuitively, condition 1 holds for all iterations. Condition 2 can be proved by induction, as the following.

At the first iteration, by lines 6 to 11, we can immediately obtain that $\mathit{workingLeaf} = \{lp \in \mathit{leaf}(p_i) \mid \mathit{Extensible}(lp, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(lp) \notin \mathit{visited}\}$ and condition 2 holds. Suppose that at the beginning of the m^{th} iteration, condition 2 holds with $\mathit{workingLeaf} = w_m$, $p_i = p_m$. After executing line 13, we can obtain that $\mathit{workingLeaf} = w_m - \{p_k\}$. By lines 15 to 21, $\mathit{wokingPrefix} = (w_m - \{p_k\}) \cup \{lp \in \mathit{leaf}(p'_k) \mid \mathit{Extensible}(lp, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(lp) \notin \mathit{visited}\}$ (1). Let $\widehat{p}_k$ be the new value of p_k after executing line 16, by the definition of $\mathit{ConcatTree}$, we have $\widehat{p}_k = (p_k, p'_k)$ and thus $\mathit{leaf}(\widehat{p}_k) = \mathit{leaf}(p'_k)$ (2). Consequently, we have $\mathit{leaf}(p_i) = (\mathit{leaf}(p_m) - \{p_k\}) \cup \mathit{leaf}(\widehat{p}_k)$, since the leaf prefix p_k has been extended to be a tree prefix $\widehat{p}_k$. Since $w_m = \{p \in \mathit{leaf}(p_m) \mid \mathit{Extensible}(p, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(p) \notin \mathit{visited}\}$, with (1) and (2), we can obtain that at the beginning of the $(m+1)^{th}$ iteration condition 2 holds. By the definition of $\mathit{Extensible}$, we can conclude that $\forall\, t \in \mathit{tasks}(p_i), \alpha \in \mathit{acts}(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in \mathit{LastT}(p_i) \wedge (\alpha \notin \mathit{safe}(\varphi) \vee \alpha \in \mathit{sd}(\mathcal{S}_i)) \Rightarrow \alpha \in \mathit{lastAct}(p_i)$ holds when the while loop terminates. Thus the Cartesian vector generated by Algorithm 2 is valid.

Further, by the definition of $\mathit{Extensible}$ we can conclude that $\forall\, t \in \mathit{tasks}(p_i), \alpha \in \mathit{acts}(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in \mathit{LastT}(p_i) \wedge (\alpha \notin \mathit{safe}(\varphi) \vee \alpha \in \sum^{sd}) \Rightarrow \alpha \in \mathit{lastAct}(p_i)$ holds when the while loop terminates. Thus the Cartesian vector generated by Algorithm 2 is valid. As for termination, we assume that all variables are finite-domain and thus the state space is finite. On one hand, the function $\mathit{GetPrefix}(\mathcal{S}, \mathcal{C}, \varphi)$ always terminates and returns a valid prefix, which has been proved in Theorem 2. On the other hand, Algorithm 2 uses $\mathit{visited}$ to store states that have been used to generate new prefixes, and by lines 7 and 18 a state is used at most once to generate a new prefix, and termination guaranteed. □

Let φ be a property and ψ be the set of propositions belonging to φ. In the following, we discuss the stuttering equivalent relation of different objects w.r.t. ψ and the notation is simplified as stuttering equivalent when ψ is clear.

Let $L(\mathcal{C})$ be the valuation of the truth values of ψ in state $\mathcal{C}$. Given two traces $\sigma = \mathcal{C}_0, \alpha_0, \mathcal{C}_1, \alpha_1, \cdots, \mathcal{C}_i, \alpha_i, \cdots$ and $\sigma' = \mathcal{C}'_0, \alpha'_0, \mathcal{C}'_1, \alpha'_1, \cdots, \mathcal{C}'_i, \alpha'_i, \cdots$, they are referred to as stuttering equivalent, i.e., $\sigma \equiv_{st_\psi} \sigma'$, iff $L(\mathcal{C}_0) = L(\mathcal{C}'_0)$ and for every integer set $M = \{m_0, m_1, \cdots, m_i\}$ ($i \geq 0$), there exists another integer set $P = \{p_0, p_1, \cdots, p_i\}$, such that $m_0 = p_0 = 0 \wedge$ for all $0 \leq k < i$, there exists $n_k, q_k > 0$ such that $m_{k+1} = m_k + n_k \wedge L(\mathcal{C}_{m_k}) = L(\mathcal{C}_{m_k+1}) = \cdots = L(\mathcal{C}_{m_k+(n_k-1)}) \neq L(\mathcal{C}_{m_{k+1}}) \wedge p_{k+1} = p_k + q_k \wedge L(\mathcal{C}_{m_k}) = L(\mathcal{C}'_{p_k}) = L(\mathcal{C}'_{p_k+1}) = \cdots = L(\mathcal{C}'_{p_k+(q_k-1)}) \wedge L(\mathcal{C}'_{p_k+1}) = L(\mathcal{C}_{m_{k+1}})$, and vice versa.

Let $exc(\mathcal{C}, \mathcal{S})$ be the set of traces obtained by executing only actions from $\mathcal{S}$ following the original semantics. Given a prefix p, we define $traces(p)$ to be the set of traces that could be obtained by p. Formally, $traces(p) = p \in LeafPrefix \wedge traces(p) = \{tr(p)\} \vee \{\sigma \mid \exists\, bp \in br(p).\ \sigma' \in traces(bp) \Rightarrow \sigma = tr(p) + \sigma'\}$, where "+" concatenates two traces. Lemma 7 illustrates that Algorithm 3 returns a prefix of traces stuttering equivalent to those generated by the original semantics. It shows that for all possible local interleaving from $\mathcal{C}$ for a certain sensor $\mathcal{S}$, the sensor prefix obtained by $GetPrefix$ contains the same sequences of valuations for the set of propositions ψ of the property φ. We refer interested readers to [1] for the proof of this lemma.

Lemma 7. *Given a state $\mathcal{C}$, let $p = GetPrefix(\mathcal{S}, \mathcal{C}, \varphi)$ be the prefix obtained by Algorithm 3. For all $\sigma \in exc(\mathcal{C}, \mathcal{S})$, there exists $\sigma' \in traces(p)$ such that $\sigma \equiv_{st_\psi} \sigma'$, and vice versa.* □

Two transition systems $\mathcal{T}$ and $\mathcal{T}'$ are said to be stuttering equivalent w.r.t. φ iff $\mathcal{C}_0 = \mathcal{C}'_0$ where $\mathcal{C}_0(\mathcal{C}'_0)$ is the initial state of $\mathcal{T}(\mathcal{T}')$, and for every trace σ in $\mathcal{T}$ there exists a trace σ' in $\mathcal{T}'$ such that $\sigma \equiv_{st_\psi} \sigma'$, and vice versa. In the following, we prove that the transition system obtained by the two-level POR approach is stuttering equivalent with the transition system obtained by the original sensor network semantics.

Theorem 4. *Given a sensor network $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$, let $\mathcal{T}$ be the transition system of $\mathcal{N}$ by the original semantics and let $\mathcal{T}'$ be the transition system obtained after applying the two-level partial order reduction w.r.t. φ over $\mathcal{N}$. Then $\mathcal{T}'$ and $\mathcal{T}$ are stuttering equivalent w.r.t. φ.*

Proof. Let ψ be the set of propositions contained in φ, and let $\mathcal{C}_0$, $\mathcal{C}'_0$ be the initial state of $\mathcal{T}$, $\mathcal{T}'$, respectively. It is immediately true that $\mathcal{C}_0 = \mathcal{C}'_0$ because both $\mathcal{T}$ and $\mathcal{T}'$ are obtained from the initial state of $\mathcal{N}$. We will prove that for any trace $\sigma = \mathcal{C}_0, \alpha_0, \cdots, \alpha_m, \mathcal{C}_{m+1}$ from $\mathcal{T}$, there exists a trace $\sigma' = \mathcal{C}'_0, \alpha'_0, \cdots, \alpha'_m, \mathcal{C}'_{m+1}$ in $\mathcal{T}'$ such that $\sigma \equiv_{st_\psi} \sigma'$. This will be proved by induction in the number of updating the valuation of ψ in a certain trace σ.

Base Case: if the number of updating the valuation of ψ in σ is zero, then we have $L(\mathcal{C}_0) = \cdots = L(\mathcal{C}_{m+1})$. Since $\mathcal{C}_0$ belongs to $\mathcal{T}'$, then let $\sigma' = \mathcal{C}'_0$ and $\sigma' \equiv_{st_\psi} \sigma$.

Induction Step: suppose that when the number of updating the valuation of ψ in σ is x, there exits $\sigma' = \mathcal{C}_0, \alpha'_0 \cdots \alpha'_n, \mathcal{C}_{m+1}$ such that $\sigma \equiv_{st_\psi} \sigma'$, and it also holds for the case when there are $(x+1)$ changes in the valuation of ψ in σ.

Let $\alpha_{k_1}, \alpha_{k_2}, \cdots, \alpha_{k_x}$ be the actions in σ such that $\alpha_{k_i} \notin safe(\varphi)$ for all $1 \leq i \leq x$. Suppose that there exist $l_1, \cdots, l_x$ such that for all $1 \leq i \leq x$, $0 \leq l_i \leq n \wedge \alpha_{k_i} \in \sum_{\mathcal{S}_{l_i}}$. Suppose that α_{k_i} is the last extendable action from a task $t_{k_i} \in Tasks(S_{l_i})$ such that $t_{k_i} \not\subset_{GI} S_{l_i}$ (1) where each k_i is ordered as follows. Given two last extendable actions $\alpha_{k_a} \in \sum_{\mathcal{S}_{l_a}}$, $\alpha_{k_b} \in \sum_{\mathcal{S}_{l_b}}$ $(l_a \neq l_b)$, if α_{k_a} is a $Send$ action and $\alpha_{k_b} \in t$, $t \not\subset_{GI} \mathcal{S}_{l_b}$ then there exists α'_{k_b} which is a receive interrupt handler in S_{l_a}. If $a < b' < b$ then $k_b < k_a$, otherwise $k_a > k_b$. The same reasoning is applied when α_{k_b} is $Send$ action and $\alpha_{k_a} \in t$, $t \not\subset_{GI} \mathcal{S}_{\alpha_{k_a}}$. If α_{k_a} and α_{k_b} are both $Send$ actions or they belong to a task $t \not\subset_{GI} \mathcal{S}_{l_b}$ then $k_a > k_b$ if $a > b$ and vice verse.

By independence of global actions two consecutive actions $\alpha_{s-1} \in \Sigma_{S_l}, \alpha_s \notin \Sigma_{S_l}$ can be permuted for all $0 \leq s \leq k_0$ and the trace $\langle \cdots, \mathcal{C}_{s-1}, \alpha_{s-1}, \mathcal{C}_s, \alpha_s,$

Table 1. Experiment Results with NesC@PAT

App (LOC / sensor)	Property	Size	#State	#Trans	Time(s)	OH(ms)	#States *wo POR*	POR Ratio
Anti-theft (3391)	Deadlock free	3	1.2M	1.2M	791	95	>2.3G	$< 6 \times 10^{-4}$
	$\Box$(theft$\Rightarrow \Diamond$alert)		1.3M	1.4M	2505	108	>4.6G	$< 3 \times 10^{-4}$
Trickle (332)	$\Diamond$AllUpdated	2	3268	3351	3	2	111683	3×10^{-2}
		3	208K	222K	74	3	>23.7M	$< 8 \times 10^{-3}$
		4	838K	947K	405	4	>5.4G	$< 2 \times 10^{-4}$
		5	13.3M	15.7M	8591	5	>1232.2G	$< 1 \times 10^{-5}$

$\mathcal{C}_{s+1}, \cdots \rangle$ is equivalent to the trace $\langle \cdots, \mathcal{C}'_{s-1}, \alpha_s, \mathcal{C}'_s, \alpha_{s+1}, \mathcal{C}'_{s+1}, \cdots \rangle$. It is possible to get a trace $\sigma_{k_0} = \mathcal{C}_0, \alpha'_0, \cdots, \alpha_{k_0}, \mathcal{C}'_{k_0}$ such that for $0 \leq j \leq k_0$, $\alpha_j \in \Sigma_{S_l}$. Let $cv = (p_0, \cdots p_l, \cdots, p_n) = GetNewCV(\mathcal{C}_0)$. By Algorithm 2, Theorem 3 and Lemma 7, there exits a trace $\sigma'_{k_0} \in traces(p_l)$ such that $\sigma'_{k_0} = \mathcal{C}_0, \alpha''_0, \cdots, \mathcal{C}''_{k_0}$ and $\sigma_{k_0} \equiv_{st_\varphi} \sigma'$. Repeating this for all k_i in (1) and by transitivity of stuttering [18] we get that $\sigma_{k_x} = \mathcal{C}_0, \cdots, \alpha_{k_0}, \cdots, \alpha_{k_x}, \mathcal{C}_{k_x+1} \equiv_{st_\varphi} \sigma'_{k_x} = \mathcal{C}_0, \cdots, \alpha'_{k_0}, \cdots, \alpha'_{k_x}, \mathcal{C}_{k_x+1}$. Permuting again $\cdots, \mathcal{C}_{s-1}, \alpha_{s-1}, \mathcal{C}_s, \alpha_s, \mathcal{C}_{s+1}, \cdots$, for all $k_x \leq s \leq k_{x+1}$ and by Algorithm 2, Theorem 3, and Lemma 7 $\sigma_i = \mathcal{C}_0, \cdots, \alpha_{k_1}, \cdots, \alpha_{k_{x+1}}, \mathcal{C}_{k_{x+1}+1} \equiv_{st_\varphi} \sigma'_i = \mathcal{C}_0, \cdots, \alpha'_{k_1}, \cdots, \alpha'_{k_{x+1}}, \mathcal{C}'_{k_{x+1}+1}$ and the number of changes in the valuations of ψ is $(x+1)$. By I.H. and transitivity of stuttering $\sigma \equiv_{st_\psi} \sigma'$. □

It has been shown that if two structures $\mathcal{T}, \mathcal{T}'$ are stuttering equivalent w.r.t. an LTL-X property φ, then $\mathcal{T}' \models \varphi$ if and only if $\mathcal{T} \models \varphi$ [6]. Therefore, our method preserves LTL-X properties.

5 Experiments and Discussion

We implemented our approach in NesC@PAT [25], a domain-specific model checker for sensor networks implemented using NesC programs. Static analysis is conducted at compile time to identify the global and local independence relations among actions and tasks, and then Algorithm 1 is adopted for state space exploration. In this section, we first evaluate the performance of the two-level POR method using a number of real-world SN applications. Then a comparison between our POR and the POR implemented in T-Check [17] is provided, since T-Check provides verification of TinyOS/NesC programs with a POR algorithm. All necessary materials for re-running the experiments can be obtained from [1].

5.1 Enhancing NesC@PAT with Two-Level POR

First, we used NesC@PAT to model check an anti-theft application and the Trickle algorithm [16]. The anti-theft application is taken from the TinyOS distribution, in which each sensor runs a NesC program of over 3000 LOC. The Trickle algorithm is widely used for code propagation in SNs, and we adopted a simplified implementation to show the reduction effects. For the anti-theft application, we checked if a sensor turns on its theft led whenever a theft is detected, i.e., $\Box$(theft $\Rightarrow \Diamond$alert). Deadlock checking was also performed for anti-theft. As for the Trickle algorithm, we checked that eventually all the nodes are updated with the latest data among the network, i.e., $\Diamond AllUpdated$.

Table 2. Comparison with T-Check

#Node	NesC@PAT					T-Check					
	wt POR			#State *wo POR*	Ratio	#Bound	*wt POR*			#State *wo POR*	Ratio
	#State	Exh	Time(s)				#State	Exh	Time(s)		
2	3012	Y	2	52.3K	6×10^{-2}	20	4765	Y	1	106.2K	$\approx 4 \times 10^{-2}$
3	120K	Y	20	>11.8M	$< 1 \times 10^{-2}$	12	66.2K	N	1	13.5M	$\approx 5 \times 10^{-3}$
						50	12.6M	Y	283	NA	NA
4	368K	Y	58	>2.7G	$< 1 \times 10^{-4}$	10	56.7K	N	1	41.8M	$\approx 1 \times 10^{-3}$
						50	420.7M	Y	1291	NA	NA
5	4.2M	Y	638	>616G	$< 7 \times 10^{-6}$	8	85.2K	N	1	17.4M	$\approx 1 \times 10^{-3}$
						50	NA	N	>12600	NA	NA

Verification results are presented in Table 1. Column OH shows the computational overhead for static analysis, which is dependent on LOC, network size and the property to be checked. This overhead is negligible (within 1 second) even for a large application like Anti-theft. The second last column estimates the complete state space size and we calculate the reduction ratio as $POR\ ratio$ ($=\frac{\#State\ wt\ POR}{\#State\ wo\ POR}$). For safety properties, $\#State\ wo\ POR$ is estimated as $S_1 \times S_2 \cdots \times S_n$, where S_i is the state space of the i^{th} sensor; as for LTL properties, it is further multiplied by the size of the Büchi automaton of the corresponding LTL property. Note that this estimation of $\#State\ wo\ POR$ is an under approximation since the state space of a single sensor is calculated without networked communication. Therefore, the $POR\ Ratio$ (both in Table 1 and 2) is also an under approximation. Therefore, our POR approach achieves a reduction of at least 10^2-10^6. Further, the larger a network is, the more reduction it will achieve.

5.2 Comparison with T-Check

In this section, we compared the performance of our POR approach and that of T-Check, by checking the same safety property for the Trickle algorithm, on the same testbed with Ubuntu 10.04 instead of Windows XP. The safety property is to guarantee that each node never performs a wrong update operation. We focused on reachability analysis as T-Check lacks support of LTL. We approximated the POR ratio obtained by T-Check by the number of states explored, i.e., $POR\ Ratio \approx \frac{\#State\ wt\ POR}{\#State\ wo\ POR}$, because T-Check adopts stateless model checking. Moreover, there is no way to calculate the complete state space of a single sensor and thus it is difficult to estimate the complete state space like what we did for NesC@PAT. Thus, we had to set small bounded numbers (around 10) in order to obtain the number of states explored by T-Check without the POR setting. The results indicate that for small networks with two or three nodes, both approaches gain similar POR ratio, but for larger networks with over four nodes, our approach outperforms T-Check significantly. We present the comparison of both approaches in Table 2, where Exh indicates if all states are explored. The POR method of T-Check treats all actions within the same sensor as $dependent$, i.e., it only reduces inter-sensor concurrency. Thus, our two-level approach would be able to obtain better reduction since intra-sensor concurrency is also minimized. Another observation is that T-Check explores more states per second, which is reasonable since T-Check does not maintain all explored states. However, our approach is more efficient in state space exploration,

taking shorter time (10^2-10^3). This is mainly because T-Check may explore the same path multiple times due to its stateless model checking.

6 Related Work

This work is related to tools/methods on exploring state space of SNs.

Approaches like SLEDE [13] and the work by McInnes [19] translate NesC programs into formal description techniques (FDT) like *Promela* (supported by SPIN) or CSP_M (supported by FDR) and use existing model checkers to conduct verification tasks. Anquiro [20] translates Conitiki C code into *Bogor* models and uses BOGOR to perform the verification. Anquiro [20] is built based on the Bogor model checking framework [21,22], for model checking WSN software written in C language for Contiki OS [8]. Source codes are firstly abstracted and converted to Anquiro-specific models, i.e., Bogor models with domain-specific extensions. Then Bogor is used to model check the models against user-specified properties. Anquiro provides three levels of abstraction to generate Anquiro-specific models and state hashing technique is adopted to reduce state space, and thus Anquiro is able to verify a network with hundreds of nodes within half an hour. However, since many low-level behaviors are abstracted away, Anquiro might not be able to detect certain bugs. Moreover, translation-based approaches could cause inaccurate results due to the semantic difference between NesC and FDTs. Hence, approaches for direct verifying NesC programs have been developed.

Werner et. al. studied the *ESAWN* protocol by producing abstract behavior models from TinyOS applications, and used CBMC to verify the models [23]. The original *ESAWN* consists of 21000 LOC, and the abstract behavior model contains 4400 LOC. Our approach is comparable to this approach, since we support SNs with thousands of LOC per sensor. Werner's work is dedicated to checking the *ESAWN* protocol and it abstracts away all platform-related behaviors. Tos2CProver [4,5] translates embedded C code to standard C to be verified by CBMC, and a POR approach is integrated. Our work differs from this work in that Tos2CProver only checks single-node TinyOS applications instead of the whole network. T-Check [17] is built on TOSSIM [15] and checks the execution of SNs by DFS or random walk to find a violation of safety properties. T-Check adopts stateless and bounded model checking and is efficient to find bugs, and it helped to reveal several unknown bugs. However, T-Check might consume a large amount of time (days or weeks) to find a violation if a large bounded number is required due to the (equivalently) complete state space exploration. T-Check applies POR at network level to reduce the state space and our approach complements it with a more effective POR which preserves LTL-X.

This work is also related to research on partial order reduction in general. Approaches that using static analysis to compute a sufficient subset of enabled actions for exploration are proposed, such as persistent/sleep set [11] and ample set [6] approaches. There are also dynamic methods which compute persistent sets of transitions on the fly [9,24]. A Cartesian POR [12] was presented to delay context switches between processes for concurrent programs.

7 Conclusions

In conclusion, we proposed a two-level POR to reduce the state space of SNs significantly, based on the independence of actions. We extended the Cartesian semantics to deal with concurrent systems with multiple levels of non-determinism such as SNs. POR was then achieved by static analysis of independence and the sensor network Cartesian semantics. We also showed that it preserves LTL-X properties. We implemented this two-level POR approach in the NesC model checker NesC@PAT and it had significantly improved the performance of verification, by allowing sensor networks with thousands of LOC in each sensor to be model checked exhaustedly, with a reduction ratio sometimes more than 10^6. One of our future directions is to apply abstraction techniques like [20] to obtain an abstracted model before applying POR to support large networks with hundreds of nodes, and another is to adopt BDD techniques to implement symbolic model checking.

References

1. Experiment Materials, http://www.comp.nus.edu.sg/~pat/NesC/por
2. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless Sensor Networks: a Survey. Computer Networks 38(4), 393–422 (2002)
3. Archer, W., Levis, P., Regehr, J.: Interface contracts for TinyOS. In: IPSN, Massachusetts, USA, pp. 158–165 (2007)
4. Bucur, D., Kwiatkowska, M.: Bug-Free Sensors: The Automatic Verification of Context-Aware TinyOS Applications. In: Tscheligi, M., de Ruyter, B., Markopoulus, P., Wichert, R., Mirlacher, T., Meschterjakov, A., Reitberger, W. (eds.) AmI 2009. LNCS, vol. 5859, pp. 101–105. Springer, Heidelberg (2009)
5. Bucur, D., Kwiatkowska, M.Z.: On software verification for sensor nodes. Journal of Systems and Software 84(10), 1693–1707 (2011)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
7. Culler, D.E., Hill, J., Buonadonna, P., Szewczyk, R., Woo, A.: A Network-Centric Approach to Embedded Software for Tiny Devices. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 114–130. Springer, Heidelberg (2001)
8. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: LCN, pp. 455–462 (2004)
9. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121. ACM (2005)
10. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: PLDI, pp. 1–11 (2003)
11. Godefroid, P., Wolper, P.: Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. Formal Methods in System Design 2(2), 149–164 (1993)
12. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian Partial-Order Reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
13. Hanna, Y., Rajan, H., Zhang, W.: SLEDE: a domain-specific verification framework for sensor network security protocol implementations. In: WISEC, pp. 109–118 (2008)
14. Levis, P., Gay, D.: TinyOS Programming, 1st edn. Cambridge University Press (2009)
15. Levis, P., Lee, N., Welsh, M., Culler, D.E.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: SenSys, pp. 126–137 (2003)

16. Levis, P., Patel, N., Culler, D.E., Shenker, S.: Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In: NSDI, California, USA, pp. 15–28 (2004)
17. Li, P., Regehr, J.: T-Check: bug finding for sensor networks. In: IPSN, Stockholm, Sweden, pp. 174–185 (2010)
18. Luttik, B., Trčka, N.: Stuttering Congruence for *Chi*. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 185–199. Springer, Heidelberg (2005)
19. McInnes, A.I.: Using CSP to Model and Analyze TinyOS Applications. In: ECBS, California, USA, pp. 79–88 (2009)
20. Mottola, L., Voigt, T., Osterlind, F., Eriksson, J., Baresi, L., Ghezzi, C.: Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In: SESENA, pp. 32–37 (2010)
21. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC/SIGSOFT FSE, pp. 267–276 (2003)
22. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: A Flexible Framework for Creating Software Model Checkers. In: TAIC PART, pp. 3–22 (2006)
23. Werner, F., Faragó, D.: Correctness of Sensor Network Applications by Software Bounded Model Checking. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 115–131. Springer, Heidelberg (2010)
24. Yang, Y., Chen, X., Gopalakrishnan, G.C., Kirby, R.M.: Efficient Stateful Dynamic Partial Order Reduction. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008)
25. Zheng, M., Sun, J., Liu, Y., Dong, J.S., Gu, Y.: Towards a Model Checker for NesC and Wireless Sensor Networks. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 372–387. Springer, Heidelberg (2011)

Compositional Sequentialization of Periodic Programs

Sagar Chaki[1], Arie Gurfinkel[1], Soonho Kong[1], and Ofer Strichman[2]

[1] CMU, Pittsburgh, USA
[2] Technion, Haifa, Israel
{chaki,arie}@cmu.edu, soonhok@cs.cmu.edu, ofers@ie.technion.ac.il

Abstract. We advance the state-of-the-art in verifying periodic programs – a commonly used form of real-time software that consists of a set of asynchronous tasks running periodically and being scheduled preemptively based on their priorities. We focus on an approach based on sequentialization (generating an equivalent sequential program) of a time-bounded periodic program. We present a new compositional form of sequentialization that improves on earlier work in terms of both scalability and completeness (i.e., false warnings) by leveraging temporal separation between jobs in the same hyper-period and across multiple hyper-periods. We also show how the new sequentialization can be further improved in the case of harmonic systems to generate sequential programs of asymptotically smaller size. Experiments indicate that our new sequentialization improves verification time by orders of magnitude compared to competing schemes.

1 Introduction

Real-Time Embedded Software (RTES) controls a wide range of safety-critical systems – ranging from airplanes and cars to infusion pumps and microwaves – that impact our daily lives. Clearly, verification of such systems is an important problem domain. A modern RTES is inherently asynchronous (since it interacts with the real world), concurrent (to increase CPU utilization allowing for smarter, smaller, and more efficient systems), and must adhere to timing constraints. Developing such a RTES is therefore quite challenging.

A common way to address this challenge is to develop the RTES not as an arbitrary concurrent system, but as a *periodic program (PP)*. Indeed, PPs are supported by many real-time OSs, including OSEK [1], vxWorks [3], and RTEMS [2]. A PP $\mathcal{C}$ consists of a set of asynchronous tasks $\{\tau_i\}_i$, where each task $\tau_i = (I_i, T_i, P_i, C_i, A_i)$ is given by a priority I_i (higher number means higher priority), a loop-free body (i.e., code) T_i, a period P_i, a worst case execution time (WCET) C_i and an arrival time A_i. Each execution of T_i is called a *job*. The least common multiple of the periods of all the tasks is called a *hyper-period*.

The execution of $\mathcal{C}$ consists of a number of threads – one per task. A *legal* execution of $\mathcal{C}$ is one in which for every i, T_i is executed by its corresponding thread exactly once between time $A_i + (k-1) \cdot P_i$ and $A_i + k \cdot P_i$, for all

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 536–554, 2013.

natural $k > 0$. A common method for achieving this goal in the real-time systems literature is to assume a preemptive fixed priority-based scheduler, and assign priorities according to the Rate Monotonic Scheduling (RMS) discipline. In RMS, shorter period implies higher priority. This is why a priority I_i is an element in the definition of τ_i. We assume that $\mathcal{C}$ is *schedulable* (i.e., it only produces legal executions) under RMS.

An example of a PP is the nxt/OSEK-based [1] LEGO Mindstorms controller (described further in Sec. 7) for a robot simulating a Turing machine. It has four periodic tasks: a *TapeMover*, with a 250ms period, that moves the tape; a *Reader*, with a 250ms period, that reads the current symbol; a *Writer*, with a 250ms period, that writes the current symbol; and a *Controller*, with a 500ms period, that issues commands to the other three tasks. Another example is a generic avionic mission system that was described in [18]. It includes 10 periodic tasks, including weapon release (10 ms), radar tracking (40 ms), target tracking (40 ms), aircraft flight data (50 ms), display (50 ms) and steering (80 ms).

The topic of this paper is verification of logical properties (i.e., user supplied assertions, race conditions, deadlocks, API usage, etc.) of periodic programs. Surprisingly, this verification problem has not received significant research attention. While a PP is a concurrent program with priorities and structured timing constraints, most recent work on concurrent verification does not support priorities or priority-locks used by such systems, which motivates a solution tailored specifically to this domain.

In our previous work [7], we presented an approach for time-bounded verification of PPs, that given a PP $\mathcal{C}$ with assertions and a time-bound W, determines whether the assertions of $\mathcal{C}$ can be violated within the time bound W. The key idea there is to use W to derive an upper bound on the number of jobs that each task can execute within W, sequentialize the resulting job-bounded concurrent program, and use an off-the-shelf sequential verifier such as CBMC [6]. We call the sequentialization used in [7] *monolithic* (monoSeq).

Compositional Sequentialization. In this paper, we develop a new *compositional* sequentialization (compSeq) that improves substantially over monoSeq in terms of both scalability and completeness (i.e., less false warnings). The compositional nature of compSeq emerges from: (i) its use of information about tasks to deduce that certain jobs are *temporally separated*, i.e., one cannot preempt the other; and (ii) using this information to restrict legal thread interleavings. In particular, compSeq leverages two types of temporal separation between jobs: (i) among jobs in the same hyper-period (intra-HP); and (ii) between jobs from different hyper-periods (inter-HP). We illustrate these concepts – and thus the key difference between compSeq and monoSeq – with an example.

Intra-HP Temporal Separation. Consider a PP $\mathcal{C} = \{\tau_0, \tau_1, \tau_2\}$, where

$$\tau_0 = (0, T_0, 100, 50, 0) \qquad \tau_1 = (1, T_1, 50, 1, 0) \qquad \tau_2 = (2, T_2, 25, 1, 0) \qquad (1)$$

That is, task τ_0 has the lowest priority (i.e., 0), body T_0, period 100, WCET 50, and arrival time of 0, and similar for the other tasks. During the time-bound

$W = 100$, there is one execution of T_0, 2 of T_1 and 4 of T_2. Hence, monoSeq constructs sequentialization of the following concurrent program:

$$T_0 \parallel (T_1 ; T_1) \parallel (T_2 ; T_2 ; T_2 ; T_2) \tag{2}$$

and adds additional constraints to remove interleavings that are infeasible due to timing and priority considerations. monoSeq ignores the arrival time information and assumes that the lowest priority task τ_0 can be preempted by any execution of any higher priority task. This leads to large number of interleavings negatively affecting both scalability and completeness. In contrast, the new approach compSeq, notes that τ_2 arrives at the same time as τ_1 and τ_0 and is therefore given the CPU first. Hence the first instance of τ_2 does not interleave with any other task. Similarly, it notes that because of the WCET, the last execution of τ_2 does not interleave with any other task either. Thus, it uses the following concurrent program

$$T_2 ; T_1 ; (T_0 \parallel (T_2 ; T_2 ; T_1)) ; T_2 \tag{3}$$

That is, the single repetition of the lowest priority task τ_0 is interleaved only with some of the executions of higher-priority tasks. In this example, it is easy to see that both (2) and (3) over-approximate $\mathcal{C}$, but (3) is more sequential, hence it leads to a sequentialization that has fewer spurious interleavings and is easier to analyze.

Inter-HP Temporal Separation. Next, suppose the time-bound W is increased to 200 (i.e., to two hyper-periods). In this case, the concurrent program constructed by monoSeq becomes:

$$(T_0 ; T_0) \parallel (T_1 ; T_1 ; T_1 ; T_1) \parallel (T_2 ; T_2 ; T_2 ; T_2 ; T_2 ; T_2 ; T_2 ; T_2) \tag{4}$$

However, note that 100 is the hyper-period of $\mathcal{C}$. It is easy to see that if all tasks initially arrive at time 0, then any execution that starts within a hyper-period ends before the end of the hyper-period. Thus, instead of monolithically encoding all execution in a given time bound, it is sufficient to encode repetitions of a hyper-period. In particular, in this example, compSeq sequentializes the following program:

$$\underbrace{T_2 ; T_1 ; (T_0 \parallel (T_2 ; T_2 ; T_1)) ; T_2)}_{\text{Sequentialization of HP\#1}} \; ; \; \underbrace{(T_2 ; T_1 ; (T_0 \parallel (T_2 ; T_2 ; T_1)) ; T_2)}_{\text{Sequentialization of HP\#2}} \tag{5}$$

Note that verifying the sequentialization of one of the two hyperperiods (HP#1 or HP#2) in isolation is not sound since they communicate via global variables.

Our experimental results show that the difference in the encoding has a dramatic effect on performance. We show that monoSeq does not scale at all on a small (but realistic) robotics controller, but compSeq is able to solve many verification problems in the order of minutes.

Contributions. The paper makes the following contributions. First, we present CompSeq when the time bound is a single hyper-period of $\mathcal{C}$, focusing on its use of intra-HP separation. Interestingly, we show that assuming that all tasks start together – a common assumption in schedulability analysis [17] – is unsound for verification (see Theorem 2). That is, a program is schedulable iff it is schedulable assuming that all tasks start at time 0. But, the program might be safe when all tasks start together, but not safe if they start at different times.

Second, we improve CompSeq for the case of *harmonic* PPs – a class of PPs in which for every pair of tasks τ_1 and τ_2, $\text{lcm}(P_1, P_2) = \max(P_1, P_2)$, i.e., P_1 is a multiple of P_2. In practice, PPs are often designed to be harmonic, for example, to achieve 100% CPU utilization [13] and more predictable battery usage [20]. The improved version, called HarmonicSeq, uses intra-HP separation just like CompSeq, but generates sequential programs of asymptotically smaller size (both theoretically and empirically).

Third, we extend CompSeq (and HarmonicSeq) to multiple hyper-periods of $\mathcal{C}$ by applying it individually to each hyper-period and composing the results sequentially, thereby leveraging inter-HP separation. We show that while this is *unsound* in general (see the discussion after Theorem 3), i.e., a periodic program is not always logically equivalent to repeating the sequentialization of its hyper-period ad infinitum, it is sound under the specific restrictions on arrival times already imposed by [7].

We have implemented our approach and validated it by verifying several RTES for controlling two flavors of LEGO Mindstorms robots – one that self-balances, avoids obstacles and responds to remote controls, and another that simulates a Turing machine. We observe that verification with CompSeq is much faster (in one case by a factor of 480x) than that with MonoSeq. The improvement is more pronounced with increasing number of hyper-periods. In many cases, verification with CompSeq completes while MonoSeq runs out of resources. Further details are presented in Sec. 7.

The rest of this paper is organized as follows. Sec. 2 discusses preliminary concepts. Sec. 3 and Sec. 4 present CompSeq and HarmonicSeq, respectively, but for one hyper-period. Sec. 5 extends them to multiple hyper-periods. In Sec. 6, we survey related work. Sec. 7 presents experimental results, and Sec. 8 concludes.

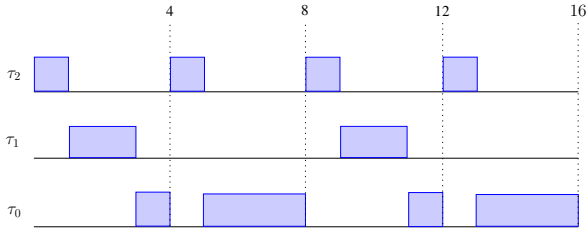
2 Preliminaries

A *task* τ is a tuple $\langle I, T, P, C, A\rangle$, where I is the priority, T – a bounded procedure (i.e., no unbounded loops or recursion) called the task body, P – the period, C – the worst case execution time (WCET) of T, and A, called the release time, is the time at which the task is first enabled[1]. A *periodic program* (PP) is a set of tasks. In this paper, we consider a N-task PP $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$, where $\tau_i = \langle I_i, T_i, P_i, C_i, A_i\rangle$. We assume that: (i) for simplicity, $I_i = i$; (ii) execution

[1] We assume that time is given in some fixed time unit (e.g., milliseconds).

Task	C_i	P_i
τ_2	1	4
τ_1	2	8
τ_0	8	16

(a)



(b)

Fig. 1. (a) Three tasks from Example 1; (b) A schedule of the three tasks

times are positive, i.e., $C_i > 0$; and (iii) priorities are *rate-monotonic* and distinct – tasks with smaller period have higher priority.

Semantics. A PP is executed by running each task periodically, starting at the release time. For $k \geq 0$ the k-th job of τ_i becomes enabled at time $A_i^k = A_i + k \times P_i$. The execution is asynchronous, preemptive, and priority-sensitive – the CPU is always given to the enabled task with the highest priority, preempting the currently executing task if necessary. Formally, the semantics of $\mathcal{C}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0\ ;\ \textbf{while}(\textsc{Wait}(\tau_i, k_i))\ (T_i\ ;\ k_i := k_i + 1) \tag{6}$$

where $\|$ is priority-sensitive interleaving, $k_i \in \mathbb{N}$ is a counter and $\textsc{Wait}(\tau_i, k_i)$ returns FALSE if the current time is greater than $A_i^{k_i}$, and otherwise blocks until time $A_i^{k_i}$ and then returns TRUE.

Schedulability. An execution of each task body T_i in (6) is called a *job*. A job's *arrival* is the time when it becomes enabled (i.e., $\textsc{Wait}(\tau_i, k)$ in (6) returns *true*); *start* and *finish* are the times when its first and last instructions are executed, respectively; *response time* is the difference between its finish and arrival times. The *response time* of task τ_i, denoted by RT_i, is the maximum response times of all of its jobs over all possible executions. Since tasks have positive execution times, their response times are also positive, i.e., $RT_i > 0$.

Note that WAIT in (6) returns TRUE if a job has finished before its next period. A periodic program is *schedulable* iff there is no run of (6) (legal with respect to priorities) in which WAIT returns FALSE. That is, a program is schedulable iff in every run every task starts and finishes within its period.

There are well-known techniques [17] to decide schedulability of periodic programs. In this paper, we are interested in logical properties of periodic programs, assuming that they meet their timing constraints. Thus, we assume that $\mathcal{C}$ is a schedulable periodic program.

Example 1. Consider the task set in Fig. 1(a). Suppose that $RT_2 = 1$, $RT_1 = 3$ and $RT_0 = 16$. A schedule demonstrating these values is shown in Fig. 1(b).

Time-Bounded Verification. Initially, in Sec. 3 and 4, we assume that $\mathcal{C}$ executes for one "hyper-period" $\mathcal{H}$. The hyper-period [17] of $\mathcal{C}$ is the least common

multiple of $\{P_0, \ldots, P_{n-1}\}$. Thus, we verify the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$ that executes like $\mathcal{C}$ for time $\mathcal{H}$ and then terminates. Subsequently, in Sec. 5, we show how to extend verification of $\mathcal{C}_{\mathcal{H}}$ to multiple hyper-periods.

Throughout the paper, we assume that the first job of each task finishes before its period, i.e.,

$$\forall 0 \leq i < N \centerdot A_i + RT_i \leq P_i \ . \tag{7}$$

Under this restriction, the number of jobs of task τ_i that executes in $\mathcal{C}_{\mathcal{H}}$ is:

$$J_i = \frac{\mathcal{H}}{P_i} \ . \tag{8}$$

The semantics of $\mathcal{C}_{\mathcal{H}}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1} \ k_i := 0 \ ; \textbf{while}(k_i < J_i \wedge \textsc{Wait}(\tau_i, k_i)) \ (T_i \ ; k_i := k_i + 1) \ . \tag{9}$$

This is analogous to the semantics of $\mathcal{C}$ in (6) except that each task τ_i executes J_i jobs. We write $\mathsf{J}(\tau, k)$ to denote the k-th job (i.e., the job at the k-th position) of task τ. Thus, the set of all jobs of $\mathcal{C}_{\mathcal{H}}$ is:

$$\mathsf{J} = \bigcup_{0 \leq i < N} \{\mathsf{J}(\tau_i, k) \mid 0 \leq k < J_i\} \ . \tag{10}$$

3 Job-Bounded Verification

We use a two-step approach to verify an N-task periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ under a time bound $\mathcal{H}$. The first step, sequentialization, outputs a non-deterministic sequential program $\mathcal{S}$ with assume statements, as shown in Algorithm 1. The second step is the verification of $\mathcal{S}$ with an off-the-shelf program verifier. In the rest of this section, we present our first sequentialization algorithm compSeq.

Sequentialization: Intuition. compSeq uses the idea that any execution π of $\mathcal{C}_{\mathcal{H}}$ can be partitioned into scheduling *rounds* in the following way: (a) π begins in round 0, and (b) a round ends and a new one begins every time a job ends (i.e., the last instruction of some task body is executed). For example, the bounded execution shown in Fig. 1(b) is partitioned into 7 rounds as follows: round 0 is the time interval $[0, 1]$ – the end of the first job of τ_2, round 1 is $[1, 3]$ – the end of the first job of τ_1, round 2 is $[3, 5]$ – the end of the second job of τ_2 (note that there is only one job of τ_0 and it ends at time 16), round 3 is $[5, 9]$, etc.

Observe that $R = |\mathsf{J}|$ jobs start and end in π, and thus π has R rounds. Therefore, compSeq reduces the bounded concurrent execution of $\mathcal{C}_{\mathcal{H}}$ into a sequential execution with R rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, each job is executed independently, in lexicographic order of increasing priority and job position. That is, lower priority jobs are executed first, and jobs of the same task are ordered by their position in the task.

In addition, compSeq leverages arrival time of each job in two important ways. First, by observing that if in every execution a job j completes before another job j' arrives, then j can precede j' in the sequentialization, independently of the priorities of the jobs. Second, by only exploring job schedules that do not violate arrival constraints. This has several benefits. First, compSeq is more complete – it generates fewer false warnings. Second, it enables eager checking for user-specified assertions incrementally.

We now present compSeq in detail. We first describe the job ordering used by compSeq, and then the sequential program $\mathcal{S}$ that compSeq generates.

Job Ordering. Consider a job $j = \mathsf{J}(\tau, k)$. Let A be the arrival time of the first job of τ and P be the period of τ. Then, the arrival time of j is $A(j) = A + k \times P$. Similarly, let RT be the response time of τ. Then the departure time of j is $D(j) = A(j) + RT$. Since we assume that $RT > 0$, we know that $A(j) < D(j)$. Let $\pi(j)$ denote the priority of its task τ. We first present three ordering relations $\lhd$, $\uparrow$ and $\sqsubset$ on jobs. Informally, $j_1 \lhd j_2$ means that j_1 always completes before j_2 begins, $j_1 \uparrow j_2$ means that it is possible for j_1 to be preempted by j_2, and $\sqsubset$ is the union of $\lhd$ and $\uparrow$.

Definition 1. *The ordering relations $\lhd$, $\uparrow$ and $\sqsubset$ are defined as follows:*

$$
\begin{aligned}
j_1 \lhd j_2 &\iff (\pi(j_1) \leq \pi(j_2) \wedge D(j_1) \leq A(j_2)) \vee (\pi(j_1) > \pi(j_2) \wedge A(j_1) \leq A(j_2)) \\
j_1 \uparrow j_2 &\iff \pi(j_1) < \pi(j_2) \wedge A(j_1) < A(j_2) < D(j_1) \\
j_1 \sqsubset j_2 &\iff A(j_1) < A(j_2) \vee (A(j_1) = A(j_2) \wedge \pi(j_1) > \pi(j_2))
\end{aligned}
$$

Lemma 1 relates $\sqsubset$ with $\lhd$ and $\uparrow$.

Lemma 1. *For any two jobs j_1 and j_2, we have:*

$$j_1 \sqsubset j_2 \iff j_1 \lhd j_2 \vee j_1 \uparrow j_2 \tag{11}$$

Note that $j_1 \sqsubset j_2$ means that either j_1 always completes before j_2, or it is possible for j_1 to be preempted by j_2. Also, $\sqsubset$ is a total strict ordering since it is a lexicographic ordering by (arrival time, priority). Moreover, $\sqsubset$ is computable in $\mathcal{O}(R \cdot \log(R))$ time, where R is the total number of jobs.

Construction of $\mathcal{S}$. The structure of $\mathcal{S}$ is given by the pseudo-code in Alg. 1. The top-level function is main. It sets (line 4) the global variables at the beginning of the first round to their initial values, and then calls hyperPeriod to execute the sequential program corresponding to a time-bound of $\mathcal{H}$.

hyperPeriod first calls scheduleJobs to create a legal job schedule – i.e., assign a starting round $start[j]$ and an ending round $end[j]$ to each job j. It then executes, in the order induced by $\sqsubset$, each job j by invoking runJob(j).

In scheduleJobs, line 12 ensures that $start[j]$ and $end[j]$ are sequential and within legal bounds; lines 13-14 ensure that jobs are properly separated; line 15 ensures that jobs are well-nested – if j_2 preempts j_1, then it finishes before j_1.

Algorithm 1. The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

```
1:  var rnd, start[], end[], localAssert[]
2:  ∀g ∈ G . var g[], v_g[]

3:  function MAIN( )
4:      ∀g ∈ G . g[0] := i_g
5:      HYPERPERIOD()

6:  function HYPERPERIOD( )
7:      SCHEDULEJOBS()
8:      ∀g ∈ G . ∀r ∈ [1, R).
            v_g[r] := *; g[r] := v_g[r]
        let the ordering of jobs by ⊏ be
        j_0 ⊏ j_1 ⊏ ... j_{R-1}
9:      RUNJOB(j_0); ...; RUNJOB(j_{R-1})

10: function SCHEDULEJOBS( )
11:     ∀j ∈ J . start[j] = *; end[j] = *
    // Jobs are sequential
12:     ∀i ∈ [0, N) . ∀k ∈ [0, J_i) . assume
            (0 ≤ start[J(i,k)] ≤ end[J(i,k)] < R)
    // Jobs are well-separated
13:     ∀j_1 ◁ j_2 . assume(end[j_1] < start[j_2])
14:     ∀j_1 ↑ j_2 . assume(start[j_1] ≤ start[j_2])
    // Jobs are well-nested
15:     ∀j_1 ↑ j_2 . assume(start[j_2] ≤ end[j_1]
            ⟹ (start[j_2] ≤ end[j_2] < end[j_1]))

16: function RUNJOB(Job j)
17:     localAssert[j] := 1
18:     rnd := start[j]
19:     T̂(j)
20:     assume(rnd = end[j])
21:     if rnd < R − 1 then
22:         ∀g ∈ G . assume
                (g[rnd] = v_g[rnd + 1])
23:     X := {j' | (j' = j ∨ j' ↑ j) ∧
            (∀j'' ≠ j . j' ↑ j'' ⇒ j'' ⊏ j)}
24:     ∀j' ∈ X . assert(localAssert[j'])

25: function T̂(Job j)
    Obtained from T_t by replacing
      each statement 'st' with:
26:     CS(j) ; st[g ← g[rnd]]
      and each 'assert(e)' with:
27:     localAssert[j] := e

28: function CS(Job j)
29:     if (*) then return FALSE
30:     o := rnd ; rnd := *
31:     assume(o < rnd ≤ end[j])
        ∀j' ∈ J . j ↑ j' ⟹
32:         assume(rnd ≤ start[j'] ∨
                rnd > end[j'])
33:     return TRUE
```

We assume, without loss of generality, that a job j contains at most one `assert`, and use the variable $localAssert[j]$ to represent the argument to this assertion. RUNJOB(j) first initializes $localAssert[j]$ to 1. It then ensures that j starts in round $start[j]$ (line 18). Next, it executes the body of j (via $\hat{T}(j)$), ensures that j terminates in round $end[j]$, and ensures consistency of round $end[j]$ by checking that the final value of g in it equals its guessed initial value in round $end[j] + 1$. Finally, it checks whether j caused an assertion violation.

Function $\hat{T}(j)$ is identical to the body of j's task, except that it uses variable $g[rnd]$ instead of g and records the argument to its assertion instead of checking it. This is important because assertions must be checked only after ensuring consistency of all relevant rounds. Function $\hat{T}(j)$ also increases value of rnd non-deterministically (by invoking function CS) to model preemption by higher priority jobs. As in other work [14], preemption is only allowed before access of global variables, without losing soundness.

Soundness of compSeq. The state-of-the-art sequentialization for periodic programs – which we refer to as monoSeq– was developed and shown to be sound in our prior work [7]. monoSeq first executes all jobs in increasing order of priority and job position, then ensures round consistency, and finally checks for assertion violations. In contrast, in the case of compSeq: (1) jobs are serialized in the order $\sqsubset$; (2) the consistency of round $end[j]$ is checked as soon as job j completes, we call this *eager-check-assumptions*; (3) assertions are also checked as soon as all jobs that affect the outcome of an assertion has completed, we call this *eager-check-assertions*. Theorem 1 states that despite these differences, the soundness of monoSeq to carry over to compSeq.

Theorem 1. *Ordering jobs by* $\sqsubset$, eager-check-assumptions, *and* eager-check-assertions *are sound.*

The set of arrival times is called the *phasing*. A special case is *zero-phasing*, when all tasks start together – i.e., $\forall i \in [0, N)\,.\ A_i = 0$. It depends on the OS whether zero-phasing can be assumed or not. For example, OSEK enforces zero-phasing while RTLinux does not.

Zero-phasing is a sufficient assumption for completeness of schedulability analysis, i.e., a system is schedulable for all phasings iff it is schedulable for the zero-phasing [17]. However, assuming zero-phasing is unsound for verification, as illustrated by Theorem 2. At the same time, assuming arbitrary phasing (as in monoSeq) leads to many false positives (see Sec. 7). compSeq handles tasks with a given phasing. This makes it sound, yet more complete than monoSeq.

Theorem 2. *The safety of a periodic program under zero-phasing does not imply its safety under all phasings.*

Proof. Consider a periodic program $\mathcal{C}$ with two tasks: t_1 and t_2. The tasks communicate via a shared variable x, initially 0. Task t_1: period 2ms, priority 1, T_1 is `assert`$(x\%2 = 1)$; task t_2: period 1ms, priority 2, T_2 is $x = x + 1$. The WCET of both tasks is 0.1ms. Under zero phasing, there is no preemption, and t_1 always reads an odd value of x. Hence the assertion succeeds. However, if t_1 arrives before t_2, then the value of x read by t_1 is 0, which fails the assertion. Therefore, $\mathcal{C}$ is safe under zero-phasing, but not under all phasings. Since $\mathcal{C}$ is harmonic, the theorem holds for harmonic periodic programs as well. □

4 Verifying Harmonic Periodic Programs

In the worst case, the number of constraints in scheduleJobs() is quadratic in the total number of jobs. This is essentially because relations $\lhd$ and $\uparrow$ collectively can have $\mathcal{O}(R^2)$ job pairs. In this section, we show that for a special class of periodic programs, known as harmonic programs, we are able to implement scheduleJobs using $\mathcal{O}(R \cdot N)$ constraints only, where N is the number of tasks. This leads to our sequentialization algorithm harmonicSeq. Since N is typically exponentially smaller than R, harmonicSeq yields an asymptotically smaller job scheduling function.

Algorithm 2. Procedure to assign legal starting and ending rounds to jobs in a harmonic program.

1: **var** $min[], max[]$ *//extra variables*

2: **function** SCHEDULEHARMONIC( )

3: $\forall j \in \mathsf{J}\,.\ start[j] = *; end[j] = *; min[j] = *; max[j] = *$

// Correctness of min and max

4: $\forall n \in \mathcal{T}\,.\ isleaf(n) \implies \text{assume}(min[n] = start[n] \wedge max[n] = end[n])$

5: $\forall n \in \mathcal{T}\,.\ \neg isleaf(n) \implies \text{assume}(min[n] = \text{MIN}(start[n], min[first(n)]))$

6: $\forall n \in \mathcal{T}\,.\ \neg isleaf(n) \implies \text{assume}(max[n] = \text{MAX}(end[n], max[last(n)]))$

// Jobs are sequential

7: $\forall n \in \mathcal{T}\,.\ \text{assume}(low(n) \leq start[n] \leq end[n] \leq high(n))$

// Jobs are well-separated

8: $\forall n \in \mathcal{T}\,.\ hasNext(n) \implies \text{assume}(max[n] < min[next(n)])$

9: $\forall j_1 \uparrow j_2\,.\ \text{assume}(start[j_1] \leq start[j_2])$

// Jobs are well-nested

10: $\forall j_1 \uparrow j_2\,.\ \text{assume}(start[j_2] \leq end[j_1] \implies (start[j_2] \leq end[j_2] < end[j_1]))$

$\mathcal{T}(n)$	= sub-tree of $\mathcal{T}$ rooted at n	$isleaf(n)$	= true iff n is a leaf node
$level(n)$	= level of node n	$size(n)$	= number of nodes in $\mathcal{T}(n)$
$id(n)$	= position of n in the DFS pre-ordering of $\mathcal{T}$	$hasNext(n)$	= true iff n is *not* the last node at level $level(n)$
$next(n)$	= node after n at level $level(n)$	$first(n)$	= first child of n
$last(n)$	= last child of n	$maxid(n)$	= $id(n) + size(n) - 1$
$low(n)$	= $id(n) - level(n)$	$high(n)$	= $maxid(n)$

Fig. 2. Functions on each node n of the job-graph

A periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ is harmonic if $\forall 0 < i < N\,.\ P_{i-1} | P_i$, where $x|y$ means that x is divisible by y. For $0 \leq i < N-1$, let $r(\tau_i) = P_i/P_{i+1}$. Note that $\mathcal{H} = P_0$ and thus $J_0 = 1$. Also, the taskset from Example 1 defines a harmonic program. Harmonicity is a common restriction imposed by real-time system designers, especially in the safety-critical domain. For example, it is possible to achieve 100% CPU utilization [13] for a harmonic program with rate monotonic scheduling.

We begin by defining the *job-tree* $\mathcal{T}$. The nodes of $\mathcal{T}$ are the jobs of $\mathcal{C}_\mathcal{H}$, and there is an edge from $j_1 = \mathsf{J}(\tau_1, p_1)$ to $j_2 = \mathsf{J}(\tau_2, p_2)$ iff $\pi(j_2) = \pi(j_1) + 1 \wedge p_2/r(\tau_1) = p_1$. Thus, the job-tree is a balanced tree of depth N rooted at $\mathsf{J}(\tau_0, 0)$ and for $0 \leq i < N-1$, each node at level i (the root is at level 0) has r_i children.

Note that since $\mathcal{C}$ is harmonic, $\uparrow$ contains $\mathcal{O}(R \cdot N)$ job pairs. This is because if $j_1 \uparrow j_2$, then j_1 must be an ancestor of j_2 in $\mathcal{T}$, and there are $\mathcal{O}(R \cdot N)$ such pairs. Moreover, all elements of $\uparrow$ can be enumerated in $\mathcal{O}(R \cdot N)$ by checking for each node j_2 of $\mathcal{T}$, and each ancestor j_1 of j_2, whether $j_1 \uparrow j_2$.

Let nodes at the same level of $\mathcal{T}$ be ordered by increasing arrival time. For each node $n \in \mathcal{T}$, we define $size(n)$, $first(n)$, $last(n)$, $id(n)$, $maxid(n)$, $level(n)$, $low(n)$ and $high(n)$ as in Fig. 2. Note that these are statically computable from

$\mathcal{T}$. Also, $maxid(n) = \text{MAX}_{k \in \mathcal{T}(n)} id(n)$, $low(n)$ is the earliest round in which job n can start, and $high(n)$ is the latest round in which job n can finish.

Since each job is a node of $\mathcal{T}$, an assignment to $start[\,]$ and $end[\,]$ is equivalent to two functions $start$ and end from nodes of $\mathcal{T}$ to values in the range $[0, R)$. This, in turn, induces the following two additional functions from $\mathcal{T}$ to $[0, R)$:

$$min(n) = \text{MIN}_{k \in \mathcal{T}(n)} start(k) \qquad max(n) = \text{MAX}_{k \in \mathcal{T}(n)} end(k)$$

The difference between HARMONICSEQ and COMPSEQ is that HARMONICSEQ uses function SCHEDULEHARMONIC – shown in Algorithm 2 – instead of SCHEDULEJOBS. The key features of SCHEDULEHARMONIC are:

- It uses two additional arrays (defined at line 1) to represent functions min and max. It adds constraints (lines 4–6) to ensure that these arrays contain appropriate values. Note that these constraints are based on the following recursive definition of min and max:

$$min(n) = \begin{cases} start(n) & \text{if } isleaf(n) \\ \text{MIN}(start(n), min(first(n))) & \text{otherwise} \end{cases}$$

$$max(n) = \begin{cases} end(n) & \text{if } isleaf(n) \\ \text{MAX}(end(n), max(last(n))) & \text{otherwise} \end{cases}$$

- It imposes stricter constraints (line 7) over $start[\,]$ and $end[\,]$, compared to SCHEDULEJOBS. Specially, it ensures that $low(n) \leq start[n] \leq end[n] \leq high(h)$ instead of $0 \leq start[n] \leq end[n] < R$.
- It uses min, max and $\uparrow$ (lines 8–9) to ensure separation. Function SCHEDULEJOBS uses $\lhd$ and $\uparrow$ instead for this purpose.
- The relation $\uparrow$ is used (line 10), as in SCHEDULEJOBS, to ensure that jobs are well-nested.

Note that the number of constraints in SCHEDULEHARMONIC is $\mathcal{O}(R \cdot N)$. Specifically, to ensure that jobs are sequential, we require $\mathcal{O}(R)$ constraints. Also, since $\uparrow$ contains $\mathcal{O}(R \cdot N)$ job pairs, specifying that jobs are well-separated and well-nested requires $\mathcal{O}(R \cdot N)$ constraints each.

5 Verification Over Multiple Hyper-Periods

In this section, we present an approach to extend job-bounded verification to the case where the time-bound is a multiple of its hyper-period. Let $\mathcal{C}$ be a schedulable periodic program with hyper-period $\mathcal{H}$ and let the time-bound for verification be $(m \times \mathcal{H})$. From (9), it follows that the semantics of $\mathcal{C}_{(m \times \mathcal{H})}$ is given by the following asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0\,;\ \textbf{while}(k_i < m \times J_i \wedge \text{WAIT}(\tau_i, k_i))\ (T_i\,;\ k_i := k_i + 1)\ . \qquad (12)$$

Let $\mathcal{C}_{\mathcal{H}}^m$ be the program that invokes function MULTIHYPER in Fig. 3(a) with argument m. In other words, $\mathcal{C}_{\mathcal{H}}^m$ executes $\mathcal{C}_{\mathcal{H}}$ sequentially m times. Since the

```
1: var rnd, start[], end[]
2: var localAssert[]
3: ∀g ∈ G. var g[], v_g[]

4: function MULTIHYPER(k)
5:     ∀g ∈ G. g[0] := i_g
6:     for i = 1 to k do
7:         HYPERPERIOD()
8:         ∀g ∈ G. g[0] := g[R − 1]
```

(a)

Task	A_i	C_i	P_i
τ_1	1.9	0.5	2
τ_2	0	0.1	1

```
int x=0; int y=0;
void T1() {
  int t=y; assert(x == t+2); y=x;
}
void T2() {x++;}
```

(b)

Fig. 3. (a) Sequentialization for multiple hyper-periods; function HYPERPERIOD is shown in Alg. 1; (b) A periodic program $\mathcal{C}$ such that $\mathcal{C}_{2\mathcal{H}}$ is not equivalent to $\mathcal{C}^2_{\mathcal{H}}$

arrival-pattern of jobs repeats every hyper-period, it is tempting to conclude that the semantics of $\mathcal{C}_{(m\times\mathcal{H})}$ is equivalent to $\mathcal{C}^m_{\mathcal{H}}$. We show that this is true under our assumption (7) from Sec. 2 on job arrivals, but is not true in general.

Theorem 3. *Let $\mathcal{C}$ be a schedulable periodic program with hyper-period $\mathcal{H}$ satisfying assumption (7) from Sec. 2. Let $\mathcal{C}_{(m\times\mathcal{H})}$ and $\mathcal{C}^m_{\mathcal{H}}$ be programs as defined above. Then, $\mathcal{C}_{(m\times\mathcal{H})}$ and $\mathcal{C}^m_{\mathcal{H}}$ are semantically equivalent with respect to safety properties for any natural number m.*

Proof. We show, by induction on m, that every execution of $\mathcal{C}_{(m\times\mathcal{H})}$ is matched by an execution of $\mathcal{C}^m_{\mathcal{H}}$, and vice versa. The base case ($m = 1$) is trivial since, by definition, $\mathcal{C}_{\mathcal{H}} = \mathcal{C}^1_{\mathcal{H}}$. For the inductive case, let $m = k + 1$ and assume that the theorem holds for $m = k$. By assumption (7), every execution of $\mathcal{C}_{k\mathcal{H}}$ terminates within time $k\mathcal{H}$. Thus, every execution t of $\mathcal{C}_{(m\times\mathcal{H})}$ is of the form $t_1 \bullet t_2$ – where $\bullet$ denotes concatenation – such that t_1 is an execution of $\mathcal{C}_{k\mathcal{H}}$ and t_2 is an execution of $\mathcal{C}_{\mathcal{H}}$. By induction, t_1 is also an execution of $\mathcal{C}^k_{\mathcal{H}}$. Therefore, t is an execution of $\mathcal{C}^{k+1}_{\mathcal{H}} = \mathcal{C}^m_{\mathcal{H}}$. The converse is proven analogously. □

To see that assumption (7) is necessary for Theorem 3, consider the periodic program $\mathcal{C}$ shown in Fig. 3(b). The tasks communicate via two shared variables `x` and `y`. Let $m = 2$. Note that $\mathcal{C}_{2\mathcal{H}}$ violates the assertion, whereas $\mathcal{C}^2_{\mathcal{H}}$ does not. In $\mathcal{C}^2_{\mathcal{H}}$, `x == y` is an inductive loop invariant: it holds initially, and is maintained since the single job of τ_1 starts after both jobs of τ_2. Therefore, the assertion is never violated since `x == y` is always true at the beginning of each hyper-period, and `x` is incremented twice by the jobs of τ_2 before the job of τ_1 begins.

However, consider the following execution in $\mathcal{C}_{2\mathcal{H}}$: (i) the τ_2 jobs from the first hyper-period set `x` to 2; (ii) the τ_1 job from the first hyper-period set `t` to 0; (iii) the second hyper-period begins; (iv) the first job of τ_2 in the second hyper-period arrives, preempts the τ_1 job and sets `x` to 3; (v) the τ_1 jobs resumes and reads the value 3 for `x`; since the value of `t` is 0, the assertion is violated. Therefore, $\mathcal{C}_{2\mathcal{H}}$ and $\mathcal{C}^2_{\mathcal{H}}$ are not equivalent with respect to safety properties. Note that $\mathcal{C}$ is harmonic, so this is true for harmonic periodic programs as well.

In summary, Theorem 3 captures the essence of inter-HP temporal separation between jobs. It allows us to verify $\mathcal{C}_{(m\times\mathcal{H})}$ by verifying $\mathcal{C}^m_{\mathcal{H}}$ instead. Since $\mathcal{C}^m_{\mathcal{H}}$ is a sequential non-deterministic program, it can be verified by any existing software model checker. Experimental results (see Sec. 7) indicate that this new way of verifying a periodic program over multiple hyper-periods is orders of magnitude faster than the state-of-the-art.

6 Related Work

There is a large body of work in verification of logical properties of both sequential and concurrent programs (see [9] for a recent survey). However, these techniques abstract away time completely, by assuming a non-deterministic scheduler model. In contrast, we use a priority-sensitive scheduler model, and abstract time partially via our job-bounded abstraction.

A number of projects [16,5] verify timed properties of systems using discrete-time [15] or real-time [4] semantics. They abstract away data- and control-flow, and verify models only. We focus on the verification of implementations of periodic programs, and do not abstract data- and control-flow.

Recently, Kidd et al. [12] have applied sequentialization as well to verify periodic programs. Their key idea is to share a single stack between all tasks and model preemptions by function calls. They do not report on an implementation. In contrast, we use a sequentialization based on rounds, and present an efficient implementation and empirical evaluation. They also use the idea that a periodic program is equivalent to unbounded repetition of its hyper-period. However, we show that this is unsound in general and provide sufficient conditions under which this is sound.

In the context of concurrent software verification, several flavors of sequentialization have been proposed and evaluated (e.g., [14,21,11,10]). Our procedure is closest to the LR [14] style. However, it differs from LR significantly, and provides a crucial advantage over LR for periodic programs [7] because it only considers schedules that respect the priorities. The key difference are in the notion of rounds, and the number of rounds required by the two techniques.

The sequentialization in this paper extends and advances the one presented in our earlier work [7]. We already pointed out the syntactic differences in Sect. 3. Semantically, compSeq is more complete than monoSeq. The reason is that compSeq imposes stronger restrictions on possible preemptions between jobs. Consider again our taskset from Example 1. compSeq ensures that $\mathsf{J}(\tau_1, 0)$ cannot be preempted by either $\mathsf{J}(\tau_2, 2)$ or $\mathsf{J}(\tau_2, 3)$. However, monoSeq only ensures that $\mathsf{J}(\tau_1, 0)$ is preempted by at most two jobs of τ_2. It allows, for example, an execution in which $\mathsf{J}(\tau_1, 0)$ is preempted by both $\mathsf{J}(\tau_2, 2)$ and $\mathsf{J}(\tau_2, 3)$. Indeed, we encountered a false warning eliminated by compSeq during our experiments (see Sec. 7). The ordering used by compSeq enables us to prune out infeasible executions and check for assertions violations more eagerly than in monoSeq. The early check of assertions leads to faster run-times, because CBMC creates straight-line programs up to the assertion, which means that they are now shorter in length.

A further advancement over [7] is HARMONICSEQ – a specialized version of sequentialization for harmonic programs, which extends naturally to multiple hyper-periods, allowing the reuse of variables across different hyper-periods.

7 Experiments

We have developed a tool called REKH which implements HARMONICSEQ and supports multiple hyper-periods. REKH is built on the same framework as REK. CIL [19] is used to parse and transform C programs and CBMC [8] is the sequential verifier. REKH takes as input C programs annotated with entry points of each task, their periods, worst case execution times, arrival times, and the time bound $\mathcal{W}$. The output is a sequential C program $\mathcal{S}$ that is then verified by CBMC. Our implementation of HARMONICSEQ includes support for locks, in exactly the same way as in MONOSEQ, as described in [7].

To compare between MONOSEQ and HARMONICSEQ, we have evaluated REKH on a set of benchmarks from prior work, and have conducted an additional case study by building and verifying a robotics controller simulating a Turing machine. In the rest of this section, we report on this experience. The tool, the experiments, and additional information about the case study, including the video of the robot and explanation of the properties verified, are available at: `http://www.andrew.cmu.edu/user/arieg/Rek`. All experiments have been performed on a AMD Opteron 2.3 GHz processor, 94 GB of main memory running Linux.

NXTway-GS Controller. The NXTway-GS controller, nxt for short, runs on nxtOSEK [1] – a real-time operating system ported to the LEGO Mindstorms platform. nxtOSEK supports programs written in C with periodic tasks and priority ceiling locks. It is the target for Embedded Coder Robot NXT – a Model-Based Design environment for using Simulink models with LEGO robots.

The basic version of the controller has 3 periodic tasks: a *balancer*, with period of 4ms, that keeps the robot upright and monitors the bluetooth link for user commands, an *obstacle*, with a period of 48ms, that monitors a sonar sensor for obstacles, and a 96ms background task that prints debug information on an LCD screen. Note that this system is harmonic. (In [7], we used a non-harmonic variant of the system). Arrival time of all tasks were set to 0 – to model the semantics of the OSEK operating system.

We verified several versions of this controller. All of the properties (i.e., assertions) verified were in the high-frequency balancer task. The balancer goes through 3 modes of execution: INIT, CALIBRATE, and CONTROL. In INIT mode all variables are initialized, and in CALIBRATE a gyroscope is calibrated. After that, balancer goes to CONTROL mode in which it iteratively reads the bluetooth link, reads the gyroscope, and sends commands to the two motors on the robot's wheels.

The results for analyzing a single hyper-period (96ms) are shown in the top part of Table 1. Experiments nxt.ok1 (nxt.bug1) check that the balancer is in a

Table 1. Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		MONOSEQ						HARMONICSEQ					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

correct (respectively, incorrect) mode at the end of the time bound. Experiment nxt.ok2 checks that the balancer is always in one of its defined modes. Experiment nxt.bug3 checks that whenever *balancer* detects an obstacle, the *balancer* responds by moving the robot. We found that since the shared variables are not protected by a lock there is a race condition that causes the *balancer* to miss a change in the state of *obstacle* for one period. Experiment nxt.ok3 is the version of the controller where the race condition has been resolved using locks. In all cases, HARMONICSEQ dramatically outperforms MONOSEQ. Furthermore, HARMONICSEQ declares the program safe for aso.bug3. Indeed, the program is safe under zero-phasing of OSEK. Note that it was flagged unsafe by MONOSEQ, a false warning, because MONOSEQ assumes arbitrary phasing.

We have also experimented with analyzing multiple hyper-periods. In the bottom part of Table 1, we show the result for 4 hyper-periods. Results for hyper-periods 2 and 3 are similar. In the case of aso.bug3 the performance improves by a factor of 40x. Furthermore, HARMONICSEQ solves all cases, while MONOSEQ times

out in 3 instances. We conclude that harmonicSeq scales better to multiple hyper-periods than monoSeq does.

Turing Machine Case Study. We built a robot simulating the Turing Machine (TM) using LEGO Mindstorms. While our robot is a toy, it is quite similar to many industrial robots such as ones used for metal stamping. Physically, the robot consists of a conveyer belt (the tape) and levers (for reading and writing the tape). The tape is built out of 16 L-shaped black bricks. Each brick represents a bit. A bit is flipped by the write lever. The light sensor, which is attached to the read head, approaches the tape and determines the value of the current bit by emitting green light and measuring the intensity of its reflection. Due to our design of the TM, it is possible for the write lever and the read head to collide. It is the controller's responsibility to avoid this collision (i.e., read head and write lever should never approach the tape together). The tape is placed on a rail and is moved left and right by the tape motor.

The implementation has four periodic tasks – `Controller`, `TapeMover`, `Reader`, and `Writer` in order of ascending priority. The `Controller` task has 500ms period and 440ms WCET. The other three tasks each have 250ms period and 10ms WCET respectively. The `Controller` task looks up a transition table, determines next operations to execute, and gives commands to the other tasks. The `TapeMover` task moves the tape to the left (or right). The `Reader` task moves the read head back and forth by rotating the read motor and reads the current bit of the tape. The `Writer` task rotates the write lever to flip a bit.

We model the motors and the color sensor to abstract away unnecessary complexity and verify properties of interest as follows:

- *Motor.* The speed of a motor is only accessed through the API functions. Motor's counter is modeled non-deterministically but respects the current speed, i.e., if the speed is positive, consecutive samplings of the counter increase monotonically. Effectively, we abstract away the acceleration.
- *Color Sensor.* The model of the color sensor returns a non-deterministic intensity value. Additionally, it maintains two variables for the mode of the sensor – one for the current mode and one for the requested one. This reflects the actual API of the sensor. The API function `set_nxtcolorsensor()` is used to request to switch the mode. The actual transition takes relatively long time (around 440ms) and is triggered by a call to `bg_nxtcolorsensor()`.

During the case study, we developed and verified the code together. We found RekH to be scalable enough for the task and useful to find many subtle errors in early development stages. Some of the more interesting properties are summarized below:

- `ctm.ok1`: When a bit is read, all the motors are stopped to avoid mismeasurement. We added `assert(R_speed==0 && W_speed==0 && T_speed==0)` in the `Reader` task to specify this property, where `R_speed`, `W_speed`, and `T_speed` represent the speed of read, write, and tape motor respectively.

Table 2. Experimental results of concurrent Turing Machine. H = # of hyper-periods, OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE, 'N' for UNSAFE, and 'U' for timeout (85,000s); Time = verification time in sec.

			monoSeq						harmonicSeq					
Name				SAT Size			S	Time		SAT Size			S	Time
	H	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
ctm.ok1	4	613	13K	121K	2,737K	8,774K	Y	44,781	7K	111K	1,063K	3,497K	Y	93.39
ctm.ok2	4	610	13K	119K	2,728K	8,738K	Y	21,804	7K	109K	1,055K	3,467K	Y	87.60
ctm.bug2	4	611	13K	118K	2,707K	8,674K	N	2,281	7K	108K	1,047K	3,441K	N	86.18
ctm.ok3	6	612	20K	222K	6,276K	20,163K	U	—	7K	171K	1,667K	5,566K	Y	243.76
ctm.bug3	6	612	20K	214K	5,914K	19,044K	N	84,625	7K	165K	1,609K	5,383K	N	248.65
ctm.ok4	8	613	29K	333K	10,390K	33,550K	U	—	7K	222K	2,178K	7,417K	Y	534.38

- `ctm.ok2`: When a bit is read, the sensor is on green-light mode. We added `assert(get_nxtcolorsensor_mode(CSENSOR) == GREEN)` in the `Reader` task to specify this property. When we request to switch to green-light mode in the `Reader` task, it sets a flag and waits until the `Controller` task runs the background process to make the transition and clear the flag.
- `ctm.bug2`: In this case, we have the same property as `ctm.ok2`. In this implementation, however, the `Reader` task does not wait for the `Controller` task to clear the flag. Since the `Reader` task has higher priority, the `Controller` task is not able to preempt and run the background process.
- `ctm.ok3`: When the writer flips a bit, the tape motor is stopped and the read head is at the safe position to avoid a collision with the read head. We added `assert(T_speed==0 && get_count(RMOTOR)<=0)` in the `Writer` task to express this property.
- `ctm.bug3`: We have assumed that the read head is stopped as soon as it arrives at the safe position (`get_count(RMOTOR)<=0`), expressed by `assert(T_speed==0 && R_speed==0)`. However, the property does not hold of our implementation due to the sampling granularity.
- `ctm.ok4`: We verified that the writer and read motors are stopped when the tape moves by checking `assert(R_speed == 0 && W_speed == 0)` in the `TapeMover` task.

Table 2 shows the experimental results of the Turing machine. For each case, the minimum hyper-period is selected for the analysis to reach the assertion in the program. For instance, `ctm.ok4` case requires at least 8 hyper-periods to check the assertion. In all cases, compSeq dramatically outperforms monoSeq. In one case - `ctm.ok1` - the performance improves by a factor of 480x.

8 Conclusion

In this paper, we deal with the problem of verifying logical properties, such as user specified assertions, race conditions, and API usage rules, of Real-Time

Embedded Systems (RTESs). We present a technique for time-bounded verification of RTES system implemented by a periodic program in C. The novelty of the technique is in *compositional* sequentialization that takes into account inter- and intra-hyper-period temporal separation between tasks. Tasks in different hyper-periods are sequentialized separately, as well as tasks that can never interleave due to their arrival and response times. This leads to a dramatic increase in scalability of the sequentialization approach while making it more complete (i.e., reducing false positives). We have implemented the approach and illustrate it on a benchmark from [7] and on an additional case study of a robotics system.

Acknowledgment. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.[2].

References

1. nxtOSEK/JSP Open Source Platform for LEGO MINDSTORMS NXT, http://lejos-osek.sf.net
2. RTEMS Operating System, http://www.rtems.com
3. VxWorks Programmer's Guide
4. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science (TCS) 126(2) (1994)
5. Braberman, V.A., Felder, M.: Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification. In: Wang, J., Lemoine, M. (eds.) ESEC/FSE 1999. LNCS, vol. 1687, pp. 494–510. Springer, Heidelberg (1999)
6. CBMC website, http://www.cprover.org/cbmc
7. Chaki, S., Gurfinkel, A., Strichman, O.: Time-Bounded Analysis of Real-Time Systems. In: Proc. of FMCAD (2011)
8. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 27(7) (2008)
10. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-Bounded Scheduling. In: Proc. of POPL (2011)

[2] NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. (DM-0000076)

11. Ghafari, N., Hu, A.J., Rakamarić, Z.: Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 227–244. Springer, Heidelberg (2010)
12. Kidd, N., Jagannathan, S., Vitek, J.: One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 245–261. Springer, Heidelberg (2010)
13. Kuo, T.-W., Mok, A.K.: Load Adjustment in Adaptive Real-Time Systems. In: Proceedings of the Real-Time Systems Symposium, RTSS 1991 (1991)
14. Lal, A., Reps, T.: 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)
15. Laroussinie, F., Markey, N., Schnoebelen, P.: Efficient timed model checking for discrete-time systems. Theoretical Computer Science (TCS) 353(1-3) (2006)
16. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1-2) (1997)
17. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM (JACM) 20(1) (1973)
18. Locke, D.C., Vogel, D.R., Lucas, L., Goodenough, J.B.: Generic Avionics Software Specification. Technical report CMU/SEI-90-TR-8-ESD-TR-90-209, Software Engineering Institute, Carnegie Mellon University (1990)
19. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
20. Rowe, A., Lakshmanan, K., Zhu, H., Rajkumar, R.: Rate-harmonized scheduling and its applicability to energy management. IEEE Trans. Industrial Informatics 6(3), 265–275 (2010)
21. 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)

Author Index