ARCoSS

LNCS 8318

Kenneth L. McMillan
Xavier Rival (Eds.)

Verification, Model Checking, and Abstract Interpretation

15th International Conference, VMCAI 2014
San Diego, CA, USA, January 2014
Proceedings

Springer

Kenneth L. McMillan Xavier Rival (Eds.)

Verification, Model Checking, and Abstract Interpretation

15th International Conference, VMCAI 2014
San Diego, CA, USA, January 19-21, 2014
Proceedings

Springer

Volume Editors

Kenneth L. McMillan
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
E-mail: kenmcmil@microsoft.com

Xavier Rival
DI - Ecole Normale Supérieure
45, rue d'Ulm
75230 Paris Cedex 05, France
E-mail: rival@di.ens.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-54012-7 e-ISBN 978-3-642-54013-4
DOI 10.1007/978-3-642-54013-4
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013957367

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 2014, the 15th International Conference on Verification, Model Checking, and Abstract Interpretation, held during January 19–21, 2013, in San Diego, co-located with POPL 2014 (the 41st 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), Philadelphia (2012), and Rome (2013).

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 received 64 submissions. Each submission was assessed by at least three Program Committee members. The committee decided to accept 25 papers. We are glad to include in the proceedings the contributions of four invited keynote speakers: Bor-Yuh Evan Chang (University of Colorado, Boulder, USA), Cynthia Dwork (Microsoft Research, USA), Prakash Panangaden (McGill University, Canada) and Thomas Wies (New York University, USA).

We would like to thank all the members of the Program Committee and all the external reviewers for their dedicated effort in evaluating and selecting the papers to be featured in these proceedings. We are also grateful to the Steering Committee for their helpful advice and support. A special thanks goes to Dave Schmidt, who managed the budget of the conference. We would also like to thank Ruzica Piskac who acted as a publicity chair. We would like to thank Suresh Jagannathan, who managed the organization of POPL 2014 as well as David Van Horn, who coordinated the co-located events of POPL.

Finally, we are also grateful to Andrei Voronkov for having set up the EasyChair system that was used to handle the submissions, the reviews, the PC discussions and the production workflow.

November 2013

Xavier Rival
Ken McMillan

Organization

Program Committee

Christel Baier	Technische Universität Dresden, Germany
Agostino Cortesi	Università Ca Foscari of Venezia, Italy
Jerome Feret	CNRS and ENS and Inria, France
Alexey Gotsman	IMDEA Software Institute, Spain
Aarti Gupta	NEC Labs, USA
Alan J. Hu	University of British Columbia, Canada
Laura Kovacs	Chalmers University of Technology, Sweden
Daniel Kroening	University of Oxford, UK
Mark Marron	Microsoft Research, USA
Isabella Mastroeni	Università di Verona, Italy
Kenneth L. McMillan	Microsoft Research, USA
Matthew Might	University of Utah, USA
David Monniaux	CNRS and University of Grenoble, France
Kedar Namjoshi	Bell Labs, USA
Peter O'Hearn	Facebook, UK
Ruzica Piskac	Yale University, USA
Sylvie Putot	CEA-LIST, France
Xavier Rival	CNRS and ENS and Inria, France
Philipp Rümmer	Uppsala University, Sweden
Sriram Sankaranarayanan	University of Colorado at Boulder, USA
Tachio Terauchi	Nagoya University, Japan
Tayssir Touili	CNRS and University of Paris Diderot-Paris7, France
Eran Yahav	Technion, Israel

Additional Reviewers

Ahrendt, Wolfgang
Blackshear, Sam
Blieberger, Johann
Bouissou, Olivier
Bucheli, Samuel
Bundala, Daniel
Cachera, David
Cerone, Andrea
D'Osualdo, Emanuele
Dan, Andrei
Dang, Thao
Delzanno, Giorgio
Dimitrov, Dimitar
Dimitrova, Rayna
Doko, Marko
Dragan, Ioan
Fahndrich, Manuel
Ferns, Norman
Garoche, Pierre-Loïc
Gerke, Michael

Gilray, Thomas
Goubault, Eric
Graf, Susanne
Guan, Nan
Gupta, Ashutosh
Horn, Alexander
Katz, Omer
Khyzha, Artem
Klebanov, Vladimir
Kloos, Johannes
Lammich, Peter
Lampka, Kai
Lauter, Christoph
Lewis, Matt
Liang, Lihao
Liang, Shuying
Lozes, Etienne
Macedo, Hugo
Midtgaard, Jan
Navarro Perez, Juan Antonio
Navas, Jorge A
Paganelli, Gabriele
Peleg, Hila
Poetzl, Daniel
Potet, Marie-Laure
Rinetzky, Noam
Schmitz, Sylvain
Schrammel, Peter
Seghir, Mohamed Nassim
Sharma, Subodh
Song, Fu
Sousa, Marcelo
Sproston, Jeremy
Subotic, Pavle
Suenaga, Kohei
Sutre, Grégoire
Tate, Ross
Tautschnig, Michael
Van Horn, David
Vedrine, Franck
Weissenbacher, Georg
Wies, Thomas
Zwirchmayr, Jakob

Keynote Talks

Minimization of Automata by Duality

Prakash Panangaden

School of Computer Science
McGill University
Montréal, Québec
Canada
prakash@cs.mcgill.ca

A remarkable algorithm — discovered by Jan Brzozowski in the early 1960s — is based on the notion of duality. In recent work by Bonchi et al. [1, 2], Bezhanishvili et al. [3] and Dinculescu et al. [4], we have shown how to understand this algorithm as a manifestation of duality. Duality is, strictly speaking, a categorical concept but is widely used and understood much more broadly. In this talk I will explain how duality can be used for the minimization of automata. It turns out to be not just applicable to ordinary automata but also to variations like weighted automata and probabilistic automata of various kinds.

References

1. Bonchi, F., Bonsangue, M.M., Rutten, J.J.M.M., Silva, A.: Brzozowski's algorithm (Co)Algebraically. In: Constable, R.L., Silva, A. (eds.) Kozen Festschrift. LNCS, vol. 7230, pp. 12–23. Springer, Heidelberg (2012)
2. Bonchi, F., Bonsangue, M.M., Hansen, H.H., Panangaden, P., Rutten, J., Silva, A.: Algebra-coalgebra duality in Brzozowski's minimization algorithm. ACM Transactions on Computational Logic (2013)
3. Bezhanishvili, N., Kupke, C., Panangaden, P.: Minimization via duality. In: Ong, L., de Queiroz, R. (eds.) WoLLIC 2012. LNCS, vol. 7456, pp. 191–205. Springer, Heidelberg (2012)
4. Dinculescu, M., Hundt, C., Panangaden, P., Pineau, J., Precup, D.: The duality of state and observation in probabilistic transition systems. In: Bezhanishvili, G., Löbner, S., Marra, V., Richter, F. (eds.) TbiLLC. LNCS, vol. 7758, pp. 206–230. Springer, Heidelberg (2013)

From Separation Logic to First-Order Logic The Complete Journey[⋆]

Thomas Wies

New York University

Abstract. Separation logic (SL) has gained widespread popularity as a formal foundation of tools that analyze and verify heap-manipulating programs. Its great asset lies in its assertion language, which can succinctly express how data structures are laid out in memory, and its discipline of local reasoning, which mimics human intuition about how to prove heap programs correct.

While the succinctness of separation logic makes it attractive for developers of program analysis tools, it also poses a challenge to automation: separation logic is a nonclassical logic that requires specialized theorem provers for discharging the generated proof obligations. SL-based tools therefore implement their own tailor-made theorem provers for this task. However, this brings its own challenges.

The analysis of real-world programs involves more than just reasoning about heap structures. The combination of linked data structures and pointer arithmetic, in particular, is pervasive in low-level system code. Other examples include the dynamic reinterpretation of memory (e.g., treating a memory region both as a linked structure and as an array) and dependencies on data stored in linked structures (e.g., sortedness constraints). To deal with such programs, existing SL tools make simplifying and (deliberately) unsound assumptions about the underlying memory model, rely on interactive help from the user, or implement incomplete extensions to allow some limited support for reasoning about other theories.

The integration of a separation logic prover into an SMT solver can address these challenges. Modern SMT solvers already implement decision procedures for many theories that are relevant in program verification, e.g., linear arithmetic, arrays, and bit-vectors. They also implement generic mechanisms for combining these theories, treating the theory solvers as independent components. These mechanisms provide guarantees about completeness and decidability.

I will present an approach that enables complete combinations of decidable separation logic fragments with other theories in an elegant way. The approach works by reducing SL assertions to first-order logic. The target of this reduction is a decidable fragment of first-order logic that fits well into the SMT framework. That is, reasoning in separation

[⋆] This work was supported in part by NSF grant CCS-1320583.

logic is handled entirely by an SMT solver. The approach enables local reasoning about heap programs via a logical encoding of the frame rule and supports complex linked data structures with list and tree-like backbones. The reduction also opens up new possibilities for invariant generation in SL-based static analysis tools (e.g., via the integration of interpolation procedures for first-order logic).

This talk is based on joint work with Ruzica Piskac, Damien Zufferey, and Nishant Totla.

Refuting Heap Reachability

Bor-Yuh Evan Chang

University of Colorado Boulder
bec@cs.colorado.edu

Abstract. Precise heap reachability information is a prerequisite for many static verification clients. However, the typical scenario is that the available heap information, computed by say an up-front points-to analysis, is not precise enough for the client of interest. This imprecise heap information in turn leads to a deluge of false alarms for the tool user to triage. Our position is to approach the false alarm problem not just by improving the up-front analysis but by also employing after-the-fact, goal-directed *refutation analyses* that yield targeted precision improvements. We have investigated refutation analysis in the context of detecting statically a class of Android memory leaks. For this client, we have found the necessity for an overall analysis capable of path-sensitive reasoning interprocedurally and with strong updates—a level of precision difficult to achieve globally in an up-front manner. Instead, our approach uses a refutation analysis that mixes highly precise, goal-directed reasoning with facts derived from the up-front analysis to prove alarms false and thus enabling effective and sound filtering of the overall list of alarms.

Table of Contents

SAT-Based Synthesis Methods for Safety Specs[⋆]

Roderick Bloem[1], Robert Könighofer[1], and Martina Seidl[2]

[1] Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
[2] Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. Automatic synthesis of hardware components from declarative specifications is an ambitious endeavor in computer aided design. Existing synthesis algorithms are often implemented with Binary Decision Diagrams (BDDs), inheriting their scalability limitations. Instead of BDDs, we propose several new methods to synthesize finite-state systems from safety specifications using decision procedures for the satisfiability of quantified and unquantified Boolean formulas (SAT-, QBF- and EPR-solvers). The presented approaches are based on computational learning, templates, or reduction to first-order logic. We also present an efficient parallelization, and optimizations to utilize reachability information and incremental solving. Finally, we compare all methods in an extensive case study. Our new methods outperform BDDs and other existing work on some classes of benchmarks, and our parallelization achieves a super-linear speedup.

Keywords: Reactive Synthesis, SAT-Solving, Quantified Boolean Formulas, Effectively Propositional Logic.

1 Introduction

Automatic synthesis is an appealing approach to construct correct reactive systems: Instead of manually developing a system and verifying it later against a formal specification, reactive synthesis algorithms can compute a *correct-by-construction* implementation of a formal specification fully automatically. Besides the construction of full systems [4], synthesis algorithms are also used in automatic debugging to compute corrections of erroneous parts of a design [29], or in program sketching, where "holes" (parts that are left blank by the designer) are filled automatically [28].

This work deals with synthesis of hardware systems from safety specifications. Safety specifications express that certain "bad things" never happen. This is an important class of specifications for two reasons. First, bounded synthesis approaches [8] can reduce synthesis from richer specifications to safety synthesis

⋆ This work was supported in part by the Austrian Science Fund (FWF) through projects RiSE (S11406-N23 and S11408-N23) and QUAINT (I774-N23), and by the European Commission through project STANCE (317753).

problems. Second, safety properties often make up the bulk of a specification, and they can be handled in a compositional manner: the safety synthesis problem can be solved before the other properties are handled [27].

One challenge for reactive synthesis is scalability. To address it, synthesis algorithms are usually symbolic, i.e., they represent states and transitions using formulas. The symbolic representations are, in turn, often implemented using Binary Decision Diagrams (BDDs), because they provide both existential and universal quantification. However, it is well known that BDDs explode in size for certain structures [2]. At the same time, algorithms and tools to decide the satisfiability of formulas became very efficient over the last decade.

In this paper, we thus propose several new approaches to use satisfiability-based methods for the synthesis of reactive systems from safety specifications. We focus on the computation of the so-called *winning region*, i.e., the states from which the specification can be fulfilled, because extracting an implementation from this winning region is then conceptually easy (but can be computationally hard). More specifically, our contributions are as follows.

1. We present a learning-based approach to compute a winning region as a Conjunctive Normal Form (CNF) formula over the state variables using a solver for Quantified Boolean Formulas (QBFs) [19].
2. We show how this method can be implemented efficiently using two incremental SAT-solvers instead of a QBF-solver, and how approximate reachability information can be used to increase the performance. We also present a parallelization that combines different variants of these learning-based approaches to achieve a super-linear speedup.
3. We present a template-based approach to compute a winning region that follows a given structure with one single QBF-solver call.
4. We also show that fixing a structure can be avoided when using a solver for Effectively Propositional Logic (EPR) [18].
5. We present extensive experimental results to compare all these methods, to each other and to previous work.

Our experiments do not reveal *the* new all-purpose synthesis algorithm. We rather conclude that different methods perform well on different benchmarks, and that our new approaches outperform existing ones significantly on some classes of benchmarks.

Related Work. A QBF-based synthesis method for safety specifications was presented in [29]. Its QBF-encoding can have deep quantifier nestings and many copies of the transition relation. In contrast, our approach uses more but potentially cheaper QBF-queries. Becker et al. [1] show how to compute all solutions to a QBF-problem with computational learning, and how to use such an ALLQBF engine for synthesis. In order to compute all losing states (from which the specification cannot be enforced) their algorithm analyzes all one-step predecessors of the unsafe states before turning to the two-step predecessors, an so on. Our learning-based synthesis method is similar, but applies learning directly to the synthesis problem. As a result, our synthesis algorithm is more "greedy". Discovered losing states are utilized immediately in the computation of new losing states, independent of the distance to the unsafe states. Besides the computation of a winning region, computational learning has also been used for extracting small circuits from a strategy [9]. The basic idea of substituting a QBF-solver with two competing SAT-solvers has already been presented in [13] and [21]. We apply this idea to our learning-based synthesis algorithm, and adapt it to make optimal use of incremental SAT-solving in our setting. Our optimizations to utilize reachability information in synthesis are based on the concept of incremental induction, as presented by Bradley for the model-checking algorithm IC3 [6]. These reachability optimizations are completely new in synthesis, to the best of our knowledge. Recently, Morgenstern et al. [21] proposed a property-directed synthesis method which is also inspired by IC3 [6]. Roughly speaking, it computes the rank (the number of steps in which the environment can enforce to reach an unsafe state) of the initial state in a lazy manner. It maintains over-approximations of states having (no more than) a certain rank. If the algorithm cannot decide the rank of a state using this information, it decides the rank of successors first. This approach is complementary to our learning-based algorithms. One fundamental difference is that [21] explores the state space starting from the initial state, while our algorithms start at the unsafe states. The main similarity is that one of our methods also uses two competing SAT-solvers instead of a QBF-solver. Templates have already been used to synthesize combinational circuits [15], loop invariants [10], repairs [16], and missing parts in programs [28]. We use this idea for synthesizing a winning region. Reducing the safety synthesis problem to EPR is also new, to the best of our knowledge.

Outline. The rest of this paper is organized as follows. Section 2 introduces basic concepts and notation, and Section 3 discusses synthesis from safety specifications in general. Our new synthesis methods are presented in Sections 4 and 5. Section 6 contains our experimental evaluation, and Section 7 concludes. An extended version [5] of this paper contains an appendix with additional proofs and experimental results.

2 Preliminaries

We assume familiarity with propositional logic, but repeat the notions important for this paper. Refer to [3] for a more gentle introduction.

Basic Notation. In propositional logic, a *literal* is a Boolean variable or its negation. A *cube* is a conjunction of literals, and a *clause* is a disjunction of literals. A formula in propositional logic is in *Conjunctive Normal Form (CNF)* if it is a conjunction of clauses. A cube describes a (potentially partial) assignment to Boolean variables: unnegated variables are true, negated ones are false. We denote vectors of variables with overlines, and corresponding cubes in bold. E.g., $\mathbf{x}$ is a cube over the variable vector $\overline{x} = (x_1, \dots, x_n)$. We treat vectors of variables like sets if the order does not matter. An $\overline{x}$-*minterm* is a cube that contains all variables of $\overline{x}$. Cube $\mathbf{x}_1$ is a *sub-cube* of $\mathbf{x}_2$, written $\mathbf{x}_1 \subseteq \mathbf{x}_2$, if the literals of $\mathbf{x}_1$ form a subset of the literals in $\mathbf{x}_2$. We use the same notation for *sub-clauses*. Let

$F(\overline{x})$ be a propositional formula over the variables $\overline{x}$, and let $\mathbf{x}$ be an $\overline{x}$-minterm. We write $\mathbf{x} \models F(\overline{x})$ to denote that the assignment $\mathbf{x}$ satisfies $F(\overline{x})$. We will omit the brackets listing variable dependencies if they are irrelevant or clear from the context (i.e., we often write F instead of $F(\overline{x})$).

Decision Procedures. A *SAT-solver* is a tool that takes a propositional formula (usually in CNF) and decides its satisfiability. Let $F(\overline{x}, \overline{y}, \ldots)$ be a propositional formula over several vectors $\overline{x}, \overline{y}, \ldots$ of Boolean variables. We write $\mathsf{sat} := \textsc{PropSat}(F)$ for a SAT-solver call. The variable sat is assigned true if and only if F is satisfiable. We write $(\mathsf{sat}, \mathbf{x}, \mathbf{y}, \ldots) := \textsc{PropSatModel}(F(\overline{x}, \overline{y}, \ldots))$ to obtain a satisfying assignment in the form of cubes $\mathbf{x}, \mathbf{y}, \ldots$ over the different variable vectors. Let $\mathbf{a}$ be a cube. We write $\mathbf{b} := \textsc{PropUnsatCore}(\mathbf{a}, F)$ to denote the extraction of an unsatisfiable core: Given that $\mathbf{a} \wedge F$ is unsatisfiable, $\mathbf{b} \subseteq \mathbf{a}$ will be a sub-cube of $\mathbf{a}$ such that $\mathbf{b} \wedge F$ is still unsatisfiable. *Quantified Boolean Formulas (QBFs)* extend propositional logic with universal ($\forall$) and existential ($\exists$) quantifiers. A QBF (in Prenex Conjunctive Normal Form) is a formula $Q_1\overline{x} \,.\, Q_2\overline{y} \ldots . \, F(\overline{x}, \overline{y}, \ldots)$, where $Q_i \in \{\forall, \exists\}$ and F is a propositional formula in CNF. Here, $Q_i\overline{x}$ is a shorthand for $Q_i x_1 \ldots Q_i x_n$ with $\overline{x} = (x_1 \ldots x_n)$. The quantifiers have their expected semantics. A *QBF-solver* takes a QBF and decides its satisfiability. We write $\mathsf{sat} := \textsc{QbfSat}(Q_1\overline{x} \,.\, Q_2\overline{y} \ldots . \, F(\overline{x}, \overline{y}, \ldots))$ or $(\mathsf{sat}, \mathbf{a}, \mathbf{b} \ldots) := \textsc{QbfSatModel}(\exists\overline{a} \,.\, \exists\overline{b} \ldots Q_1\overline{x} \,.\, Q_2\overline{y} \ldots F(\overline{a}, \overline{b}, \ldots, \overline{x}, \overline{y}, \ldots))$ to denote calls to a QBF-solver. Note that $\textsc{QbfSatModel}$ only extracts assignments for variables that are quantified existentially on the outermost level.

Transition Systems. A *controllable finite-state transition system* is a tuple $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I, T)$, where $\overline{x}$ is a vector of Boolean state variables, $\overline{i}$ is a vector of uncontrollable input variables, $\overline{c}$ is a vector of controllable input variables, $I(\overline{x})$ is an initial condition, and $T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ is a transition relation with $\overline{x}'$ denoting the next-state copy of $\overline{x}$. A *state* of $\mathcal{S}$ is an assignment to the $\overline{x}$-variables, usually represented as $\overline{x}$-minterm $\mathbf{x}$. A formula $F(\overline{x})$ represents the set of all states $\mathbf{x}$ for which $\mathbf{x} \models F(\overline{x})$. Priming a formula F to obtain F' means that all variables in the formula are primed, i.e., replaced by their next-state copy. An *execution* of $\mathcal{S}$ is an infinite sequence $\mathbf{x}_0, \mathbf{x}_1 \ldots$ of states such that $\mathbf{x}_0 \models I$ and for all pairs $(\mathbf{x}_j, \mathbf{x}_{j+1})$ there exist some input assignment $\mathbf{i}_j, \mathbf{c}_j$ such that $\mathbf{x}_j \wedge \mathbf{i}_j \wedge \mathbf{c}_j \wedge \mathbf{x}'_{j+1} \models T$. A state $\mathbf{x}$ is *reachable* in $\mathcal{S}$ if there exists an execution $\mathbf{x}_0, \mathbf{x}_1 \ldots$ and an index j such that $\mathbf{x} = \mathbf{x}_j$. The execution of $\mathcal{S}$ is controlled by two *players*: the *protagonist* and the *antagonist*. In every step j, the antagonist first chooses an assignment $\mathbf{i}_j$ to the uncontrollable inputs $\overline{i}$. Next, the protagonist picks an assignment $\mathbf{c}_j$ to the controllable inputs $\overline{c}$. The transition relation T then computes the next state $\mathbf{x}_{j+1}$. This is repeated indefinitely. We assume that T is *complete* and *deterministic*, i.e., for every state and input assignment, there exists exactly one successor state. More formally, we have that $\forall\overline{x}, \overline{i}, \overline{c} \,.\, \exists\overline{x}' \,.\, T$ and $\forall\overline{x}, \overline{i}, \overline{c}, \overline{x_1}', \overline{x_2}' \,.(T(\overline{x}, \overline{i}, \overline{c}, \overline{x_1}') \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x_2}')) \Rightarrow (\overline{x_1}' = \overline{x_2}')$. Let $F(\overline{x})$ be a formula representing a certain set of states. The mixed pre-image $\mathsf{Force}^p_1(F) = \forall\overline{i} \,.\, \exists\overline{c}, \overline{x}' \,.\, T \wedge F'$ represents all states from which the protagonist can enforce to reach a state of F in exactly one step. Analogously, $\mathsf{Force}^a_1(F) = \exists\overline{i} \,.\, \forall\overline{c} \,.\, \exists\overline{x}' \,.\, T \wedge$

F' gives all states from which the antagonist can enforce to visit F in one step.

Synthesis Problem. A (memoryless) *controller* for $\mathcal{S}$ is a function $f : 2^{\overline{x}} \times 2^{\overline{i}} \to 2^{\overline{c}}$ to define the control signals $\overline{c}$ based on the current state of $\mathcal{S}$ and the uncontrollable inputs $\overline{i}$. Let $P(\overline{x})$ be a formula characterizing the set of safe states in a transition system $\mathcal{S}$. An execution $\mathbf{x}_0, \mathbf{x}_1 \ldots$ is *safe* if it visits only safe states, i.e., $\mathbf{x}_j \models P$ for all j. A controller f for $\mathcal{S}$ is *safe* if all executions of $\mathcal{S}$ are safe, given that the control signals are computed by f. Formally, f is safe if there exists no sequence of pairs $(\mathbf{x}_0, \mathbf{i}_0), (\mathbf{x}_1, \mathbf{i}_1), \ldots$ such that (a) $\mathbf{x}_0 \models I$, (b) $\mathbf{x}_j \wedge \mathbf{i}_j \wedge f(\mathbf{x}_j, \mathbf{i}_j) \wedge \mathbf{x}'_{j+1} \models T$ for all $j \geq 0$, and (c) $\mathbf{x}_j \not\models P$ for some j. The problem addressed in this paper is to synthesize such a safe controller. We call a pair $(\mathcal{S}, P)$ a *specification* of a safety synthesis problem. A specification is *realizable* if a safe controller exists. A *safe implementation* $\mathcal{I}$ of a specification $(\mathcal{S}, P)$ with $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I(\overline{x}), T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'))$ is a transition system $\mathcal{I} = (\overline{x}, \overline{i}, \emptyset, I(\overline{x}), T(\overline{x}, \overline{i}, f(\overline{x}, \overline{i}), \overline{x}'))$, where f is a safe controller for $\mathcal{S}$.

3 Synthesis from Safety Specifications

This paper presents several approaches for synthesizing a safe controller for a fine-state transition system $\mathcal{S}$. The synthesis problem can be seen as a game between the protagonist controlling the $\overline{c}$-variables and the antagonist controlling the $\overline{i}$-variables during an execution [21]. The protagonist wins the game if the execution never visits an unsafe state $\mathbf{x} \not\models P$. Otherwise, the antagonist wins. A safe controller for $\mathcal{S}$ is now simply a strategy for the protagonist to win the game. Standard game-based synthesis methods can be used to compute such a winning strategy [30]. These game-based methods usually work in two steps. First, a so-called *winning region* is computed. A winning region is a set of states $W(\overline{x})$ from which a winning strategy for the protagonist exists. Second, a winning strategy is derived from (intermediate results in the computation of) the winning region. Most of the synthesis approaches presented in the following implement this two-step procedure. For safety synthesis problems, the following three conditions are sufficient for a winning region $W(\overline{x})$ to be turned into a winning strategy.

I) Every initial state is in the winning region: $I \Rightarrow W$.
II) The winning region contains only safe states: $W \Rightarrow P$.
III) The protagonist can enforce to stay in the winning region: $W \Rightarrow \mathsf{Force}_1^p(W)$.

A specification is realizable if and only if such a winning region exists. Hence, it suffices to search for a formula that satisfies these three constraints. Deriving a winning strategy $f : 2^{\overline{x}} \times 2^{\overline{i}} \to 2^{\overline{c}}$ from such a winning region is then conceptually easy: f must always pick control signal values such that the successor state is in W again. This is always possible due to (I) and (III). We therefore focus on approaches to efficiently compute a winning region that satisfies (I)-(III), and leave an investigation of methods for the extraction of a concrete controller to future work[1]. First, we will briefly discuss an attractor-based approach which is often implemented with BDDs [30]. Then, we will present several new ideas which are more suitable for an implementation using SAT- and QBF-solvers.

3.1 Standard Attractor-Based Synthesis Approach

The synthesis method presented in this section can be seen as the standard textbook method for solving safety games [30]. Starting with all safe states P, the SafeSynth algorithm reduces F to states from which the protagonist can enforce to go back to F until F does not change anymore. If an initial state is removed from F, false is returned to signal unrealizability. Otherwise, F will finally converge to a fixpoint, which is a proper winning region W ($W = \nu F.P \wedge \mathsf{Force}_1^p(F)$ in μ-calculus notation). SafeSynth is well suited for an implementation using BDDs because the set of all states satisfying $\mathsf{Force}_1^p(F)$ can be computed with just a few BDD operations, and the comparison to decide if F changed can be done in constant time. A straightforward implementation using a QBF-solver maintains a growing quantified formula to represent F (i.e, $F_0 = P$, $F_1 = \exists \overline{x} \,.\, \forall \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,.\, P \wedge T \wedge P'$, and so on), and calls a QBF-solver to decide if F changed semantically from one iteration to the next one. This approach is explained in [29]. In iteration n, F contains n copies of the transition relation and $2n$ quantifier alternations. This means that the difficulty of the QBF queries increases significantly with the number of iterations, which may be prohibitive for large specification. The resulting winning region W is a quantified formula as well. An alternative QBF-based implementation [1] eliminates the quantifiers from F in every iteration by computing all satisfying assignments of F. The next section explains how this idea can be improved.

```
1: procedure SafeSynth($\mathcal{S}, P$), returns: $W$ or false
2:   $F := P$
3:   while $F$ changes do
4:     $F := F \wedge \mathsf{Force}_1^p(F)$
5:     if $I \not\Rightarrow F$ then
6:       return false
7:   return $F$
```

4 Learning-Based Synthesis Approaches

Becker et al. [1] show how SafeSynth can be implemented with a QBF-solver by eliminating the quantifiers in F with computational learning. This gives a CNF representation of every F-iterate. However, we are only interested in the final value W of F. This allows for a tighter and more efficient integration of the learning approach with the SafeSynth algorithm.

4.1 Learning-Based Synthesis Using a QBF-Solver

The following algorithm uses computational learning to compute a winning region in CNF using a QBF-solver. It returns false in case of unrealizability.

[1] In our implementation, we currently extract circuits by computing Skolem functions for the $\overline{c}$ signals in $\forall \overline{x}, \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,.(\neg W) \vee (T \wedge W')$ using the QBFCert [22] framework. However, there are other options like learning [9], interpolation [14], or templates [15].

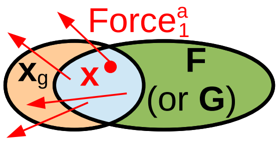


Fig. 1. LEARNQBF: working principle

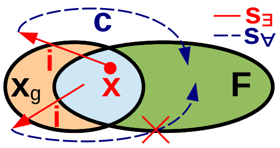


Fig. 2. LEARNSAT: working principle

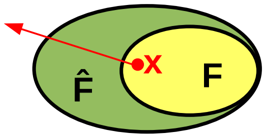


Fig. 3. LEARNSAT: Using $\hat{F}$ for incremental solving

```
1: procedure LearnQbf($(\overline{x}, \overline{i}, \overline{c}, I, T), P$), returns: $W$ or false
2:   $F := P$
3:   // Check if there exists an $\mathbf{x} \models F \wedge \mathsf{Force}_1^a(\neg F)$:
4:   while sat with $(\mathsf{sat}, \mathbf{x})$:=QbfSatModel($\exists \overline{x}, \overline{i} \,.\, \forall \overline{c} \,.\, \exists \overline{x}' \,.\, F \wedge T \wedge \neg F'$) do
5:     // Find a sub-cube $\mathbf{x}_g \subseteq \mathbf{x}$ such that $(\mathbf{x}_g \wedge F) \Rightarrow \mathsf{Force}_1^a(\neg F)$:
6:     $\mathbf{x}_g := \mathbf{x}$
7:     for $l \in$ literals($\mathbf{x}$) do
8:       $\mathbf{x}_t := \mathbf{x}_g \setminus \{l\}$, if optimize then $G := F \wedge \neg \mathbf{x}_g$ else $G := F$
9:       if $\neg$QbfSat($\exists \overline{x} \,.\, \forall \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,.\, \mathbf{x}_t \wedge G \wedge T \wedge G'$) then
10:        $\mathbf{x}_g := \mathbf{x}_t$
11:    if PropSat($\mathbf{x}_g \wedge I$) then return false
12:    $F := F \wedge \neg \mathbf{x}_g$
13:  return $F$
14: end procedure
```

The working principle of LEARNQBF is illustrated in Figure 1. It starts with the initial guess F that the winning region contains all safe states P. Line 4 then checks for a counterexample to the correctness of this guess in form of a state $\mathbf{x} \models F \wedge \mathsf{Force}_1^a(\neg F)$ from which the antagonist can enforce to leave F. Assume that $\mathsf{optimize} = \mathsf{false}$ in line 8 for now, i.e., G is always just F. The inner loop now generalizes the state-cube $\mathbf{x}$ to $\mathbf{x}_g \subseteq \mathbf{x}$ by dropping literals as long as $\mathbf{x}_g$ does not contain a single state from which the protagonist can enforce to stay in F. During and after the execution of the inner loop, $\mathbf{x}_g$ contains only states that must be removed from F, or have already been removed from F before. Hence, as an optimization, we can treat the states of $\mathbf{x}_g$ as if they were removed from F already *during* the cube minimization. This is done with $\mathsf{optimize} = \mathsf{true}$ in line 8 by setting $G = F \wedge \neg \mathbf{x}_g$ instead of $G = F$. This optimization can lead to smaller cubes and less iterations. If the final cube $\mathbf{x}_g$ contains an initial state, the algorithm signals unrealizability by returning false. Otherwise, it removes the states of $\mathbf{x}_g$ from F by adding the clause $\neg \mathbf{x}_g$, and continues by checking for other counterexamples. If P is in CNF, then the final result in F will also be in CNF. If T is also in CNF, then the query of line 9 can be constructed by merging clause sets. Only for the query in line 4, a CNF encoding of $\neg F'$ is necessary. This can be achieved, e.g., using a Plaisted-Greenbaum transformation [23], which causes only a linear blow-up of the formula.

Heuristics. We observed that the generalization (the inner loop of LEARNQBF) is often fast compared to the computation of counterexamples in Line 4. As a heuristic, we therefore propose to compute not only one but all (or several) minimal generalizations $\mathbf{x}_g \subseteq \mathbf{x}$ to every counterexample-state $\mathbf{x}$, e.g., using a hitting set tree algorithm [24]. Another observation is that newly discovered clauses can render earlier clauses redundant in F. In every iteration, we therefore "compress" F by removing clauses that are implied by others. This can be done cheaply with incremental SAT-solving, and simplifies the CNF for $\neg F'$ in line 4. Iterating over existing clauses and trying to minimize them further at a later point in time did not lead to significant improvements in our experiments.

4.2 Learning-Based Synthesis Using SAT-Solvers

LearnQbf can also be implemented with SAT-solving instead of QBF-solving. The basic idea is to use two competing SAT-solvers for the two different quantifier types, as done in [13]. However, we interweave this concept with the synthesis algorithm to better utilize incremental solving capabilities of modern SAT-solvers.

```
1: procedure LearnSat($(\overline{x}, \overline{i}, \overline{c}, I, T), P$), returns: $W$ or false
2:   $F := P$, $\hat{F} := P$, $U :=$ true, precise := true
3:   while true do
4:     $(\mathsf{sat}, \mathbf{x}, \mathbf{i}) :=$ PropSatModel($F \wedge U \wedge T \wedge \neg\hat{F}'$)
5:     if $\neg$sat then
6:       if precise then return $F$
7:       $U :=$ true, $\hat{F} := F$, precise := true
8:     else
9:       $(\mathsf{sat}, \mathbf{c}) :=$ PropSatModel($F \wedge \mathbf{x} \wedge \mathbf{i} \wedge T \wedge F'$)
10:      if $\neg$sat then
11:        $\mathbf{x}_g :=$ PropUnsatCore($\mathbf{x}, F \wedge \mathbf{i} \wedge T \wedge F'$)
12:        if PropSat($\mathbf{x}_g \wedge I$) then return false
13:        $F := F \wedge \neg\mathbf{x}_g$
14:        if optimize then precise := false else $\hat{F} := F$, $U :=$ true
15:      else
16:        $U := U \wedge \neg$PropUnsatCore($\mathbf{x} \wedge \mathbf{i}, \mathbf{c} \wedge F \wedge U \wedge T \wedge \neg\hat{F}'$)
17: end procedure
```

Data Structures. Besides the current guess F of the winning region W, LearnSat also maintains a copy $\hat{F}$ of F that is updated only lazily. This allows for better utilization of incremental SAT-solving, and will be explained below. The flag precise indicates if $\hat{F} = F$. The variable U stores a CNF formula over the $\overline{x}$ and $\overline{i}$ variables. Intuitively, U contains state-input combinations which are not useful for the antagonist when trying to break out of F.

Working Principle. The working principle of LearnSat is illustrated in Figure 2. For the moment, let optimize be false, i.e., $\hat{F}$ is always F. To deal with the mixed quantification inherent in synthesis, LearnSat uses two competing SAT-solvers, $s_\exists$ and $s_\forall$. In line 4, $s_\exists$ tries to find a possibility for the antagonist to leave F. It is computed as a state-input pair $(\mathbf{x}, \mathbf{i})$ for which some $\overline{c}$-value leads to a $\neg F$ successor. Next, in line 9, $s_\forall$ searches for a response $\mathbf{c}$ of the protagonist to avoid leaving F. If no such response exists, then $\mathbf{x}$ must be excluded from F. However, instead of excluding this one state only, we generalize the state-cube $\mathbf{x}$ by dropping literals to obtain $\mathbf{x}_g$, representing a larger region of states for which input $\mathbf{i}$ can be used by the antagonist to enforce leaving F. This is done by computing the unsatisfiable core with respect to the literals of $\mathbf{x}$ in line 11. Otherwise, if $s_\forall$ finds a response $\mathbf{c}$, then the state-input pair $(\mathbf{x}, \mathbf{i})$ is not helpful for the antagonist to break out of F. It must be removed from U to avoid that the same pair is tried again. Instead of removing just $(\mathbf{x}, \mathbf{i})$, we generalize it again by dropping literals as long as the control value $\mathbf{c}$ prevents leaving F. This is done by computing an unsatisfiable core over the literals in $\mathbf{x} \wedge \mathbf{i}$ in line 16.

As soon as F changes, U must be reset to true (line 14): even if a state-input pair is not helpful for breaking out of F, it may be helpful for breaking out of a smaller F. If line 4 reports unsatisfiability, then the antagonist cannot enforce to leave F, i.e., F is a winning region ($\mathsf{precise} = \mathsf{true}$ if $\mathsf{optimize} = \mathsf{false}$). If an initial state is removed from F, then the specification is unrealizable (line 12).

Using $\hat{F}$ to Support Incremental Solving. Now consider the case where $\mathsf{optimize}$ is true. In line 13, new clauses are added only to F but not to $\hat{F}$. This ensures that $F \Rightarrow \hat{F}$, but F can be strictly stronger. See Figure 3 for an illustration. Line 4 now searches for a transition (respecting U) from F to $\neg\hat{F}$. If such a transition is found, then it also leads from F to $\neg F$. However, if no such transition from F to $\neg\hat{F}$ exists, then this does not mean that there is no transition from F to $\neg F$. Hence, in case of unsatisfiability, we update $\hat{F}$ to F and store the fact that $\hat{F}$ is now accurate by setting $\mathsf{precise} = \mathsf{true}$. If the call in line 4 reports unsatisfiability with $\mathsf{precise} = \mathsf{true}$, then there is definitely no way for the antagonist to leave F and the computation of F is done. The reason for not updating $\hat{F}$ immediately is that solver $s_\exists$ can be used incrementally until the next update, because new clauses are only added to F and U. Only when reaching line 7, a new incremental session has to be started. This optimization proved to be very beneficial in our experiments. Solver $s_\forall$ can be used incrementally throughout the entire algorithm anyway, because F gets updated with new clauses only.

4.3 Utilizing Unreachable States

This section presents an optimization of LearnQbf to utilize (un)reachability information. It works analogously for LearnSat, though. Recall that the variable G in LearnQbf stores the current over-approximation of the winning region W (cf. Section. 4.1). LearnQbf generalizes a counterexample-state $\mathbf{x}$ to a region $\mathbf{x}_g$ such that $G \wedge \mathbf{x}_g \Rightarrow \mathsf{Force}_1^a(\neg G)$, i.e., $G \wedge \mathbf{x}_g$ contains only states from which the antagonist can enforce to leave G. Let $R(\overline{x})$ be an over-approximation of the states reachable in $\mathcal{S}$. That is, R contains at least all states that could appear in an execution of $\mathcal{S}$. It is sufficient to ensure $G \wedge \mathbf{x}_g \wedge R \Rightarrow \mathsf{Force}_1^a(\neg G)$ because unreachable states can be excluded from G even if they are winning for the protagonist. This can lead to smaller cubes and faster convergence.

There exist various methods to compute reachable states, both precisely and as over-approximation [20]. The current over-approximation G of the winning region W can also be used: Given that the specification is realizable (we will discuss the unrealizable case below), the protagonist will enforce that W is never left. Hence, at any point in time, G is itself an over-approximation of the reachable states, not necessarily in $\mathcal{S}$, but definitely in the final implementation $\mathcal{I}$ (given that $\mathcal{I}$ is derived from W and $W \Rightarrow G$). Hence, stronger reachability information can be obtained by considering only transitions that remain in G.

In our optimization, we do not explicitly compute an over-approximation of the reachable states, but rather exploit ideas from the property directed reachability algorithm IC3 [6]: By induction, we know that a state $\mathbf{x}$ is definitely unreachable in $\mathcal{I}$ if $\mathbf{x} \not\models I$ and $\neg\mathbf{x} \wedge G \wedge T \Rightarrow \neg\mathbf{x}'$. Otherwise, $\mathbf{x}$ could be reachable. The same holds for sets of states. By adding these two constraints, we modify the generalization check in line 9 of LearnQbf to

$$\begin{aligned}
&\textsc{QbfSat}(\exists \overline{x}^*, \overline{i}^*, \overline{c}^* \,.\, \exists \overline{x} \,.\, \forall \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,. \\
&\qquad (I(\overline{x}) \vee G(\overline{x}^*) \wedge \neg\mathbf{x}_g(\overline{x}^*) \wedge T(\overline{x}^*, \overline{i}^*, \overline{c}^*, \overline{x})) \wedge \\
&\qquad \mathbf{x}_g(\overline{x}) \wedge G(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')).
\end{aligned} \tag{1}$$

We will refer to this modification as optimization RG (which is short for "reachability during generalization"). Only the second line is new. Here, $\overline{x}^*$, $\overline{i}^*$, and $\overline{c}^*$ are the previous-state copies of $\overline{x}$, $\overline{i}$, and $\overline{c}$, respectively. Originally, the formula was true if the region $\mathbf{x}_g \wedge G$ contained a state from which the protagonist could enforce to stay in G. In this case, the generalization failed, because we cannot safely remove states that are potentially winning for the protagonist. The new formula is true only if $\mathbf{x}_g \wedge G$ contains a state $\mathbf{x}_a$ from which the protagonist can enforce to stay in G, and this state $\mathbf{x}_a$ is either initial, or has a predecessor $\mathbf{x}_b$ in $G \wedge \neg\mathbf{x}_g$. This situation is illustrated in Figure 4. States that are neither initial nor have a predecessor in $G \wedge \neg\mathbf{x}_g$ are unreachable and, hence, can safely be removed. Note that we require $\mathbf{x}_b$ to be in $G \wedge \neg\mathbf{x}_g$, and not just in G and different from $\mathbf{x}_a$. The intuitive reason is that a predecessor in $G \wedge \mathbf{x}_g$ does not count because this region is going to be removed from G. A more formal argument is given by the following theorem.

Theorem 1. *For a realizable specification, if Eq. 1 is unsatisfiable, then $G \wedge \mathbf{x}_g$ cannot contain a state $\mathbf{x}_a$ from which (a) the protagonist can enforce to visit G in one step, and (b) which is reachable in any implementation $\mathcal{I}$ derived from a winning region $W \Rightarrow G$ with $W \Rightarrow \mathsf{Force}_1^p(W)$.*

A proof can be found in [5]. Theorem 1 ensures that the states removed with optimization RG cannot be necessary for the protagonist to win the game, i.e., that the optimization does not remove "too much". So far, we assumed realizability. However, optimization RG also cannot make an unrealizable specification be identified as realizable. It can only remove more states, which means that unrealizability is detected only earlier.

Similar to improving the generalization of counterexamples using unreachability information, we can also restrict their computation to potentially reachable

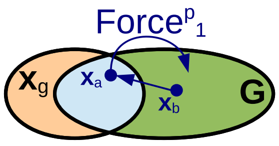


Fig. 4. Optimization RG: A counterexample to generalization

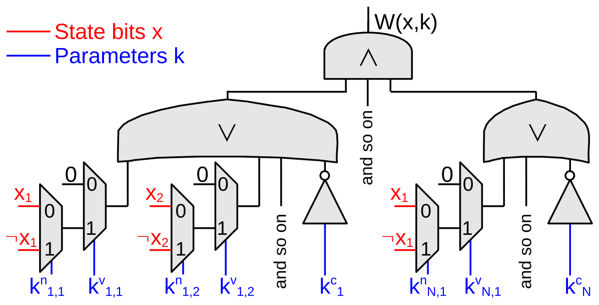


Fig. 5. A CNF template for the winning region

states. This is explained as optimization RC in [5]. However, while optimization RG resulted in significant performance gains (more than an order of magnitude for some benchmarks; see the columns SM and SGM in Table 3 of [5]), we could not achieve solid improvements with optimization RC. Sometimes the computation became slightly faster, sometimes slower.

4.4 Parallelization

The algorithms LearnQbf and LearnSat compute clauses that refine the current over-approximation F of the winning region. This can also be done with multiple threads in parallel using a global clause database F. Different threads can implement different methods to compute new clauses, or generalize existing ones. They notify each other whenever they add a (new or smaller) clause to F so that all other threads can continue to work with the refined F.

In our implementation, we experimented with different thread combinations. If two threads are available, we let them both execute LearnSat with optimization RG but without RC. We keep the LearnSat-threads synchronized in the sense that they all use the same $\hat{F}$. If one thread restarts solver $s_\exists$ with a new $\hat{F}$, then all other LearnSat-threads restart their $s_\exists$-solver with the same $\hat{F}$ as well. This way, the LearnSat-threads can not only exchange new F-clauses, but also new U-clauses. We use different SAT-solvers in the different threads (currently our implementation supports Lingeling, Minisat, and PicoSat). This reduces the chances that the threads find the same (or similar) counterexamples and generalizations. Also, the solvers may complement each other: if one gets stuck for a while on a hard problem, the other one may still achieve significant progress in the meantime. The stuck solver then benefits from this progress in the next step. We also let the LearnSat-threads store the computed counterexample-cubes in a global counterexample-database. If three threads are available, we use one thread to take counterexample-cubes from this database, and compute all possible generalizations using a SAT-solver and a hitting set tree algorithm [24].

We also experimentally added threads that minimize existing clauses further using a QBF-solver, and threads implementing LearnQbf. However, we observed that threads using QBF-solvers can not quite keep up with the pace of threads using SAT-solvers. Consequently, they only yield minor speedups.

Our parallelization approach does not only exploit hardware parallelism, it is also a playground for combining different methods and solvers. We only tried a few options; a thorough investigation of beneficial combinations remains to be done.

5 Direct Synthesis Methods

This section presents completely different approaches for computing a winning region. Instead of refining an initial guess in many iterations, we simply assert the constraints for a proper winning region and compute a solution in one go.

5.1 Template-Based Synthesis Approach

We define a generic template $W(\overline{x}, \overline{k})$ for the winning region $W(\overline{x})$, where $\overline{k}$ is a vector of Boolean variables acting as template parameters. Concrete values $\mathbf{k}$ for the parameters $\overline{k}$ instantiate a concrete formula $W(\overline{x})$ over the state variables $\overline{x}$. This reduces the search for a Boolean formula (the winning region) to a search for Boolean parameter values. We can now find a winning region that satisfies the three desired properties (I)-(III) with a single QBF-solver call:

$$\begin{aligned}(sat, \mathbf{k}) = \textsc{QbfSatModel}(\exists \overline{k} \,.\, \forall \overline{x}, \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,.\; & (I \Rightarrow W(\overline{x}, \overline{k})) \wedge \\ & (W(\overline{x}, \overline{k}) \Rightarrow P) \wedge \\ & (W(\overline{x}, \overline{k}) \Rightarrow (T \wedge W(\overline{x}', \overline{k})))\end{aligned} \tag{2}$$

The challenge in this approach is to define a generic template $W(\overline{x}, \overline{k})$ for the winning region. Figure 5 illustrates how a CNF template could look like. Here, $W(\overline{x})$ is a conjunction of clauses over the state variables $\overline{x}$. Template parameters $\overline{k}$ define the shape of the clauses. First, we fix a maximum number N of clauses in the CNF. Then, we introduce three vectors of template parameters: $\overline{k^c}$, $\overline{k^v}$, and $\overline{k^n}$. We denote their union by $\overline{k}$. If parameter k_i^c with $1 \leq i \leq N$ is true, then clause i is used in $W(\overline{x})$, otherwise not. If parameter $k_{i,j}^v$ with $1 \leq i \leq N$ and $1 \leq j \leq |\overline{x}|$ is true, then the state variable $x_j \in \overline{x}$ appears in clause i of $W(\overline{x})$, otherwise not. Finally, if parameter $k_{i,j}^n$ is true, then x_j can appear in clause i only negated, otherwise only unnegated. If $k_{i,j}^v$ is false, then $k_{i,j}^n$ is irrelevant. This gives $|\overline{k}| = 2 \cdot N \cdot |\overline{x}| + N$ template parameters. Figure 5 illustrates this definition of $W(\overline{x}, \overline{k})$ as a circuit. A CNF encoding of this circuit to be used in the QBF query shown in Eq. 2 is straightforward. Choosing N is delicate. If N is too low, we will not find a solution, even if one exists. If it is too high, we waste computational resources and may find an unnecessarily complex winning region. In our implementation, we solve this dilemma by starting with $N = 1$ and doubling it upon failure. We stop if we get a negative answer for $N \geq 2^{|\overline{x}|}$

(because any Boolean formula over $\overline{x}$ can be represented in a CNF with $< 2^{|\overline{x}|}$ clauses). The CNF template explained in this paragraph is just an example. Other ideas include And-Inverter Graphs with parameterized interconnects, or other parameterized circuits [15].

The template-based approach can be good at finding simple winning regions quickly. There may be many different winning regions that satisfy the conditions (I)-(III). The algorithms SafeSynth, LearnQbf and LearnSat will always find the largest of these sets (modulo unreachable states, if used with optimization RG or RC). The template-based approach is more flexible. As an extreme example, suppose that there is only one initial state, it is safe, and the protagonist can enforce to stay in this state. Suppose further that the largest winning region is complicated. The template-based approach may find $W = I$ quickly, while the other approaches may take ages to compute the largest winning region. On the other hand, the template-based approach can be expected to scale poorly if no simple winning region exists, or if the synthesis problem is even unrealizable. The issue of detecting unrealizability can be tackled just like in bounded synthesis [11]: in parallel to searching for a winning region for the protagonist, one can also try to find a winning region for the antagonist (a set of states from which the antagonist can enforce to leave the safe states in some number of steps). If a winning region for the antagonist contains an initial state, unrealizability is detected.

5.2 EPR Reduction Approach

The EPR approach is based on the observation that a winning region $W(\overline{x})$ satisfying the three requirements (I)-(III) can also be computed as a Skolem function, without a need to fix a template. However, the requirement (III) concerns not only W but also its next-state copy W'. Hence, we need a Skolem function for the winning region and its next-state copy, and the two functions must be consistent. This cannot be formulated as a QBF problem with a linear quantifier structure, but only using so-called Henkin Quantifiers[2] [12], or in the *Effectively Propositional Logic (EPR)* [18] fragment of first-order logic. Deciding the satisfiability of formulas with Henkin Quantifiers is NEXPTIME-complete, and only a few tools exist to tackle the problem [12]. Hence, we focus on reductions to EPR. EPR is a subset of first-order logic that contains formulas of the form $\exists \overline{A} \,.\, \forall \overline{B} \,.\, \varphi$, where A and B are disjoint vectors of variables ranging over some domain $\mathbb{D}$, and φ is a function-free first-order formula in CNF. The formula φ can contain predicates, which are (implicitly) existentially quantified.

Recall that we need to find a formula $W(\overline{x})$ such that $\forall \overline{x}, \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,.(I \Rightarrow W) \wedge (W \Rightarrow P) \wedge (W \Rightarrow T \wedge W')$. In order to get a corresponding EPR formula, we must (a) encode the Boolean variables using first-order domain variables, (b) eliminate the existential quantification inside the universal one, and (c) encode

[2] A winning region is a Skolem function for the Boolean variable w in the formula $\begin{matrix}\forall \overline{x} \,.\, \exists w \,.\, \forall \overline{i} \,.\, \exists \overline{c} \,. \\ \forall \overline{x}' \,.\, \exists w' \,.\end{matrix} \; (I \Rightarrow w) \wedge (w \Rightarrow P) \wedge ((\overline{x} = \overline{x}') \Rightarrow (w = w')) \wedge (w \wedge T \Rightarrow w')$.

the body of the formula in CNF. Just like [26], we can address (a) by introducing a new domain variable Y for every Boolean variable y, a unary predicate p to encode the truth value of variables, constants $\top$ and $\bot$ to encode true and false, and the axioms $p(\top)$ and $\neg p(\bot)$. The existential quantification of the $\overline{x}'$ variables can be turned into a universal one by turning the conjunction with T into an implication, i.e., re-write $\forall \overline{x}, \overline{i} \,.\, \exists \overline{c}, \overline{x}' \,.\, W(\overline{x}) \Rightarrow T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge W(\overline{x}')$ to $\forall \overline{x}, \overline{i} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, W(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \Rightarrow W(\overline{x}')$. This works because we assume that T is both deterministic and complete. We Skolemize the $\overline{c}$-variables $c_1, \ldots, c_n$ by introducing new predicates $C_1(\overline{X}, \overline{I}), \ldots, C_n(\overline{X}, \overline{I})$. For W, we also introduce a new predicate $W(\overline{X})$. This gives

$$\forall \overline{X}, \overline{I}, \overline{X}' \,.\; (I(\overline{X}) \Rightarrow W(\overline{X})) \quad \wedge \quad (W(\overline{X}) \Rightarrow P(\overline{X})) \quad \wedge$$
$$(W(X) \wedge T(\overline{X}, \overline{I}, \overline{C}(\overline{X}, \overline{I}), \overline{X}') \Rightarrow W(X'))$$

The body of this formula has to be encoded in CNF, but many first-order theorem provers and EPR solvers can do this internally. If temporary variables are introduced in the course of a CNF encoding, then they have to be Skolemized with corresponding predicates. Instantiation-based EPR-solvers like iProver [17] can not only decide the satisfiability of EPR formulas, but also compute models in form of concrete formulas for the predicates. For our problem, this means that we cannot only directly extract a winning region but also implementations for the control signals from the $C_j(\overline{X}, \overline{I})$-predicates. iProver also won the EPR track of the Automated Theorem Proving System Competition in the last years.

6 Experimental Results

This section presents our implementation, benchmarks and experimental results.

6.1 Implementation

We implemented the synthesis methods presented in this paper in a prototype tool. The source code (written in C++), more extensive experimental results, and the scripts to reproduce them are available for download[3]. Our tool takes as input an AIGER[4] file, defined as for the safety track of the hardware synthesis competition, but with the inputs separated into controllable and uncontrollable ones. It outputs the synthesized implementation in AIGER format as well. Several back-ends implement different methods to compute a winning region. At the moment, they all use QBFCert [22] to extract the final implementation. However, in this paper, we evaluate the winning region computation only. Table 1 describes some of our implementations. Results for more configurations (with different optimizations, solvers, etc.) can be found in the downloadable archive. The BDD-based method is actually implemented in a separate tool[5]. It uses

[3] `www.iaik.tugraz.at/content/research/design_verification/demiurge/`.
[4] See `http://fmv.jku.at/aiger/`.

Table 1. Overview of our Implementations

Name	Techn.	Solver	Description
BDD	BDDs	CuDD	SafeSynth (Sect. 3.1)
PDM	SAT	Minisat	Property directed method [21]
QAGB	QBF	BloqqerM + DepQBF	LearnQbf + opt. RG + comp. of all counterexample generalizations (Sect. 4.1)
SM	SAT	Minisat	LearnSat (Sect. 4.2)
SGM	SAT	Minisat	Like SM but with optimization RG
Pi	SAT	various	Multi-threaded with i threads (Sect. 4.4)
TB	QBF	BloqqerM + DepQBF	CNF-template-based (Sect. 3.1)
EPR	EPR	iProver	EPR-based (Sect. 5.2)

dynamic variable reordering, forced re-orderings at certain points, and a cache to speedup the construction of the transition relation. PDM is a re-implementation of [21]. These two implementations serve as baseline for our comparison. The other methods are implemented as described above. BloqqerM refers to an extension of the QBF-preprocessor Bloqqer to preserve satisfying assignments. This extension is presented in [25].

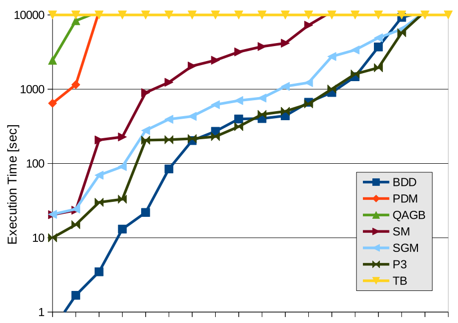
6.2 Benchmarks

We evaluate the methods on several parametrized specifications. The first one defines an arbiter for ARM's AMBA AHB bus [4]. It is parametrized with the number of masters it can handle. These specifications are denoted as `amba`ij, where i is the number of masters, and $j \in \{\texttt{c}, \texttt{b}\}$ indicates how the fairness properties in the original formulation of the specification were transformed into safety properties (see [5] for details). The second specification is denoted by `genbuf`ij, with $j \in \{\texttt{c}, \texttt{b}\}$, and defines a generalized buffer [4] connecting i senders to two receivers. Also here, liveness properties have been reduced to safety properties. Both of these specifications can be considered as "control-intensive", i.e., contain complicated constraints on few signals. In contrast to that, the following specifications are more "data-intensive", and do not contain transformed liveness properties. The specification `add`io with $o \in \{\texttt{y}, \texttt{n}\}$ denotes a combinational i-bit adder. Here o=`y` indicates that the AIGER file was optimized with ABC [7], and o=`n` means that this optimization was skipped. Next, `mult`i denotes a combinational i-bit multiplier. The benchmark `cnt`io denotes an i-bit counter that must not reach its maximum value, which can be prevented by setting the control signals correctly at some other counter value. Finally, `bs`io denotes an i-bit barrel shifter that is controlled by some signals. The tables 2 and 3 in the extended version of this paper [5] list the size of these benchmarks.

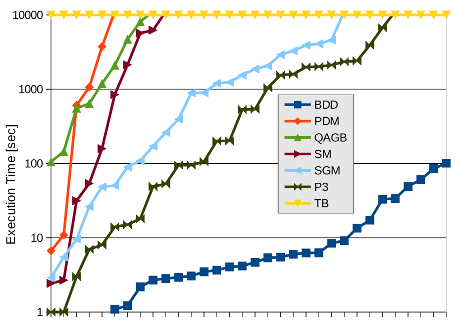
6.3 Results

Figure 6 summarizes the performance results of our synthesis methods on the different parameterized specifications with cactus plots. The vertical axis shows

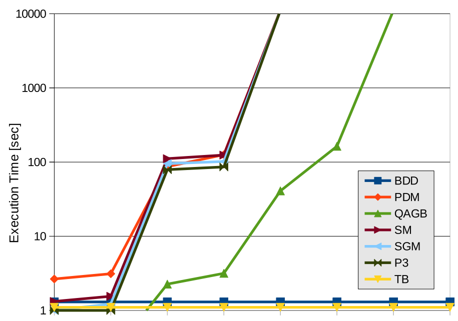
[5] Is was created by students and won a competition in a lecture on synthesis.



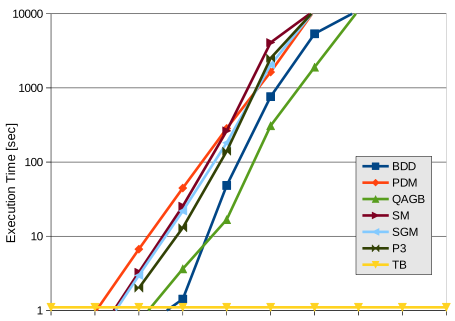
(a) Results for `amba`



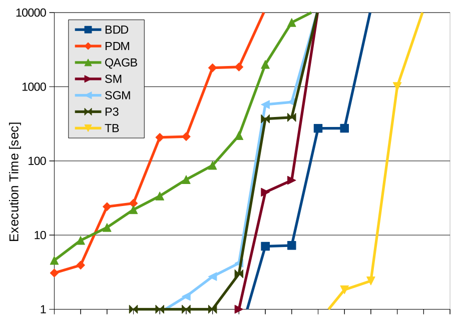
(b) Results for `genbuf`



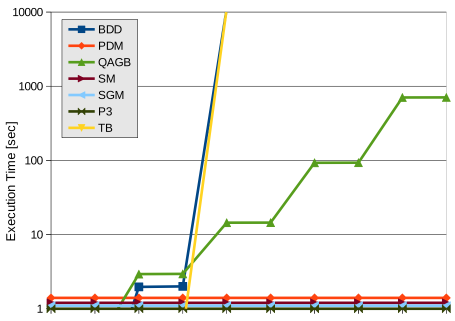
(c) Results for `add`



(d) Results for `mult`



(e) Results for `cnt`



(f) Results for `bs`

Fig. 6. Cactus plots summarizing our performance evaluation

the execution time for computing a winning region using a logarithmic scale. The horizontal axis gives the number of benchmark instances that can be solved within this time limit (per instance). Roughly speaking this means that the steeper a line rises, the worse is the scalability of this method. In order to make the charts more legible, we sometimes "zoomed" in on the interesting parts. That is, in some charts we omitted the leftmost part were all methods terminate within fractions of a second, as well as the rightmost part where (almost) all methods timeout. We set a timeout of 10 000 seconds, and a memory limit of 4 GB. The memory limit was only exceeded by the EPR approach. The EPR approach did so for quite small instances already, so we did not include it in Figure 6. The detailed execution times can be found in the tables 2 and 3 of [5]. All experiments were performed on an Intel Xeon E5430 CPU with 4 cores running at 2.66 GHz, and a 64 bit Linux. Figure 7 illustrates the speedup achieved by our parallelization (see Section 4.4) on the `amba` and `genbuf` benchmarks in a scatter plot. The x-axis carries the computation time with one thread. The y-axis shows the corresponding execution time with two and three threads. Note that the scale on both axes is logarithmic.

6.4 Discussion

Figure 7 illustrates a parallelization speedup mostly between a factor of 2 and 37, with a tendency to greater improvements for larger benchmarks. Only part of the speedup is due to the exploitation of hardware parallelism. Most of the speedup actually stems from the fact that the threads in our parallelization execute different methods and use different solvers that complement each other. Even if executed on a single CPU core in a pseudo-parallel manner, a significant speedup can be observed. In our parallelization, we experimented with only a few combinations of solvers and algorithms. We think that there is still a lot of room for improvements, requiring a more extensive investigation of beneficial algorithm and solver combinations.

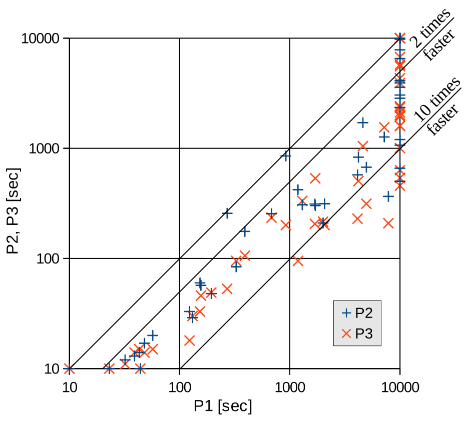


Fig. 7. Parallelization speedup

For the `amba` benchmarks, our parallelization P3 slightly outperforms BDDs (Figure 6(a)). For `genbuf`, BDDs are significantly faster (Figure 6(b)). The template-based approach does not scale at all for these benchmarks. The reason is that, most likely, no simple CNF representation of a winning region exists for these benchmarks. For instance, for the smallest `genbuf` instance, P3 computes a winning region as a CNF formula with 124 clauses and 995 literal occurrences. By dropping literals and clauses as long as this does not change the shape of the winning region, we can simplify this CNF to 111 clauses and 849 literal occurrences. These numbers indicates that no winning region for these benchmarks can be described with only a few clauses. Instantiating a CNF template with more than 100 clauses is far beyond the capabilities of the solver, because the number of template parameters grows so large (e.g., 4300 template parameters for the smallest `genbuf` instance with a template of 100 clauses for the winning region). The situation is different for `add` and `mult`. These designs are mostly combinational (with a few states to track if an error occurred). A simple

CNF-representation of the winning region (with no more than 2 clauses) exists, and the template-based approach finds it quickly (Figure 6(c) and 6(d)).

In Figure 6(b), we observe a great improvement due to the reachability optimization RG (SM vs. SGM). In some plots, this improvement is not so significant, but optimization RG never slows down the computation significantly. Similar observations can be made for QAGB (but this is not shown in the plots to keep them simple).

The SAT-based back-end SGM outperforms the QBF-based back-end QAGB on most benchmark classes (all except for `add` and `mult`). It has already been observed before that solving QBF-problems with plain SAT-solvers can be beneficial [13, 21]. Our experiments confirm these observations. One possible reason is that SAT-solvers can be used incrementally, and they can compute unsatisfiable cores. These features are missing in modern QBF-solvers. However, this situation may change in the future.

The barrel shifters `bs` are intractable for BDDs, even for rather small sizes. Already when building the BDD for the transition relation, the approach times out because of many and long reordering phases, or runs out of memory if reordering is disabled. In contrast, almost all our SAT- and QBF-based approaches are done within fractions of a second on these examples. We can consider the `bs`-benchmark as an example of a design with complex data-path elements. BDDs often fail to represent such elements efficiently. In contrast, the SAT- and QBF-based methods can represent them easily in CNF. At the same time, the SAT- and QBF-solvers seem to be smart enough to consider the complex data-path elements only as far as they are relevant for the synthesis problem.

On most of the benchmarks, especially `amba` and `genbuf`, our new synthesis methods outperform our re-implementation of [21] (PDM in Figure 6) by orders of magnitude. Yet, [21] reports impressive results for these benchmarks: the synthesis time is below 10 seconds even for `amba`16 and `genbuf`16. We believe that this is due to a different formulation of the benchmarks. We translated the benchmarks, exactly as used in [21], into our input language manually, at least for `amba`16 and `genbuf`16. Our PDM back-end, as well as most of the other back-ends, solve them in a second. This suggests that the enormous runtime differences stem from differences in the benchmarks, and not in the implementation. An investigation of the exact differences in the benchmarks remains to be done.

In summary, none of the approaches is consistently superior. Instead, the different benchmark classes favor different methods. BDDs perform well on many benchmarks, but are outperformed by our new methods on some classes. The template-based approach and the parallelization of the SAT-based approach seem particularly promising. The reduction to EPR turned out to scale poorly.

7 Summary and Conclusion

In this paper, we presented various novel SAT- and QBF-based methods to synthesize finite-state systems from safety specifications. We started with a learning-based method that can be implemented with a QBF-solver. Next, we proposed an efficient implementation using a SAT-solver, an optimization using reachability information, and an efficient parallelization that achieves a super-linear speedup by combining different methods and solvers. Complementary to that, we also presented synthesis methods based on templates or reduction to EPR. From our extensive case study, we conclude that these new methods can complement BDD-based approaches, and outperform other existing work [21] by orders of magnitude.

In the future, we plan to fine-tune our optimizations and heuristics using larger benchmark sets. We also plan to research and compare different methods for the extraction of circuits from the winning region.

Acknowledgments. We thank Aaron R. Bradley for fruitful discussions about using IC3-concepts in synthesis, Andreas Morgenstern for his support in re-implementing [21] and translating benchmarks, Bettina Könighofer also for providing benchmarks, and Fabian Tschiatschek and Mario Werner for their BDD-based synthesis tool.

References

1. Becker, B., Ehlers, R., Lewis, M., Marin, P.: ALLQBF solving by computational learning. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 370–384. Springer, Heidelberg (2012)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. FAIA, vol. 185. IOS Press (2009)
4. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. Electronic Notes in Theoretical Computer Science 190(4), 3–16 (2007)
5. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. CoRR, abs/1311.3530 (2013), `http://arxiv.org/abs/1311.3530`
6. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
7. 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)
8. Ehlers, R.: Symbolic bounded synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 365–379. Springer, Heidelberg (2010)
9. Ehlers, R., Könighofer, R., Hofferek, G.: Symbolically synthesizing small circuits. In: FMCAD 2012, pp. 91–100. IEEE (2012)
10. 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. Sci. Comput. Program. 69(1-3), 35–45 (2007)
11. Filiot, E., Jin, N., Raskin, J.-F.: An antichain algorithm for LTL realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)

12. Fröhlich, A., Kovasznai, G., Biere, A.: A DPLL algorithm for solving DQBF. In: Pragmatics of SAT (PoS 2012, aff. to SAT 2012) (2012)
13. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 230–244. Springer, Heidelberg (2011)
14. Jiang, J.-H.R., Lin, H.-P., Hung, W.-L.: Interpolating functions from large boolean relations. In: International Conference on Computer-Aided Design (ICCAD 2009), pp. 779–784. IEEE (2009)
15. Kojevnikov, A., Kulikov, A.S., Yaroslavtsev, G.: Finding efficient circuits using SAT-solvers. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 32–44. Springer, Heidelberg (2009)
16. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: FMCAD 2011, pp. 91–100. IEEE (2011)
17. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
18. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. 21(3), 317–353 (1980)
19. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. JSAT 7(2-3), 71–76 (2010)
20. Moon, I., Kukula, J.H., Shiple, T.R., Somenzi, F.: Least fixpoint approximations for reachability analysis. In: ICCAD 1999, pp. 41–44. IEEE (1999)
21. Morgenstern, A., Gesell, M., Schneider, K.: Solving games using incremental induction. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 177–191. Springer, Heidelberg (2013)
22. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF (tool presentation). In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 430–435. Springer, Heidelberg (2012)
23. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. 2(3), 293–304 (1986)
24. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)
25. Seidl, M., Könighofer, R.: Partial witnesses from preprocessed quantified Boolean formulas. In: DATE 2014 (to appear, 2014)
26. Seidl, M., Lonsing, F., Biere, A.: qbf2epr: A tool for generating EPR formulas from QBF. In: Workshop on Practical Aspects of Automated Reasoning (2012)
27. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD 2009, pp. 77–84. IEEE (2009)
28. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009)
29. Staber, S., Bloem, R.: Fault localization and correction with QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 355–368. Springer, Heidelberg (2007)
30. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)

Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs

Martin D. Schwarz[1], Helmut Seidl[1], Vesal Vojdani[2], and Kalmer Apinis[1]

[1] Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{schwmart,seidl,apinis}@in.tum.de
[2] Deptartment of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
vesal@cs.ut.ee

Abstract. Although priority scheduling in concurrent programs provides a clean way of synchronization, developers still additionally rely on hand-crafted schemes based on integer variables to protect critical sections. We identify a set of sufficient conditions for variables to serve this purpose. We provide efficient methods to verify these conditions, which enable us to construct an enhanced analysis of mutual exclusion in interrupt-driven concurrent programs. All our algorithms are build upon off-the-shelf inter-procedural analyses alone. We have implemented this approach for the analysis of automotive controllers, and demonstrate that it results in a major improvement in the precision of data race detection compared to purely priority-based techniques.

1 Introduction

Embedded computing is omnipresent in the automotive industry. Dedicated operating systems and standards, such as Autosar/OSEK[1, 11], have been created and are used by many car manufacturers. These operating systems provide sophisticated synchronization primitives, such as priority-driven scheduling and resource acquisition. Still, developers sometimes rely on hand-crafted mechanisms for ensuring safe concurrent execution, e.g., for synchronizing two cooperating interrupts that do not affect any further interrupts. One such mechanism is to use global program variables as *flags* whose values control the access to critical sections. Accordingly, any analysis of OSEK programs, such as [15], which only takes resources and priorities into account, will produce a large number of false alarms on real-world automotive code.

An example of a typical flag-based synchronization pattern used by developers is shown in Fig. 1. This program consists of two interrupt service routines that are executed by a priority driven scheduler. The low-priority interrupt I sets a flag f to 1 before entering its critical section and resets it to 0 afterwards, whereas the higher-priority interrupt Q first checks whether the flag f has been set and only enters its critical section if f equals 0. This ensures, in a priority-driven single-core concurrency setting, that the accesses to x will always be

```
int f = 0;                         /* Priority 1 */
int x = 0;                         isr_I () {
/* Priority 3 */                     f = 1;
isr_Q() {                            x++; /* Ix */
  if (f == 0)                        f = 0;
    x--; /* Qx */                  }
}
```

Fig. 1. Example code of program with a flag *f*

exclusive. Note that priorities are crucial for such non-symmetric idioms: The higher-priority interrupt Q may safely assume that it cannot itself be preempted by the lower-priority interrupt I between checking the flag and accessing the data. Conversely, having checked the flag, it would be redundant for the high priority interrupt to set and reset the flag. Idioms like this, when properly used and implemented, can indeed protect critical sections. Still, being hand-crafted, they are error-prone and may be rendered insufficient if further priority levels are introduced. If, for example, another interrupt R is introduced whose priority exceeds that of Q, but uses the same pattern to access the variable x, a potential race condition arises between interrupts Q and R.

Our goal in this paper therefore is to provide practical methods for identifying and analyzing a wide range of synchronization patterns based on global variables, in order to provide a more accurate data-race analysis.

In principle, interrupt-driven programs with priority-based scheduling on a single-core can be analyzed by interpreting interrupts as function calls possibly occurring at every program point [4, 15]. Practically, though, context- and flow-sensitive handling of the global program state is prohibitively expensive. The approach outlined in [15] is to arrive at a practical analysis of mutual exclusion for interrupt-driven programs by analyzing local data such as dynamic priorities and resources context-sensitively, while summarizing the global state into a single flow- and context-insensitive invariant. However, when global variables are used as flags, that approach is insufficient in precision.

The key contributions of this paper are, first, to identify general properties of global variables used as flags in a wide range of hand-crafted synchronization patterns, and second, using these properties to construct efficient dedicated analysis methods based on off-the-shelf inter-procedural analysis. In particular, the resulting analysis turns out to be *interrupt-modular*, meaning that each interrupt can be analyzed independently. In a second stage, the resulting summaries are combined to precisely and efficiently compute the set of values that a flag variable may take at any given program point. Finally, this information is exploited to ensure that two program points are not part of a data race.

In this paper we first consider a class of primitive flag-based synchronization patterns which allow a low-priority interrupt to protect its critical sections against cooperating interrupts from higher priority levels. For example, the synchronization pattern used in Fig. 1 falls into this class. For this restricted class of flags, it suffices to only consider the intra-interrupt value-sets of the flag variable. For that, we verify that program points can only be part of a data race if the protecting flag variable has overlapping value sets. In a second step we generalize the conditions on flag variables in order to handle a richer class of synchronization patterns, while keeping interrupt-modularity. An example for such a pattern is provided in Fig. 2 (Section 5.1).

A critical feature of both techniques is that they are sound on all programs; that is, they do not merely assume that the flags are well-behaved. Instead, they also verify the assumptions on the flag variables themselves. Thus, the techniques can be applied one after the other on-demand to ensure race freedom when purely resource-based analyses fail. We have implemented the core technique in the Goblint analyzer [18]. This not only lead to a decrease in the number of warnings on a suite of toy examples, but also resulted in a drastic reduction of alarms in our key industrial benchmarks.

2 OSEK Model

An OSEK program consists of tasks and interrupts. Tasks are activated by direct calls or timers. Interrupts can occur at any time. The priority ceiling protocol [16] is used for scheduling. Tasks and interrupts have a static initial priority, which may be dynamically increased during execution by acquiring resources and possibly decreased again by releasing resources. The dynamic priority never drops below the static initial priority. An interrupt will preempt the running execution if its static priority exceeds both the static and dynamic priority of the running task or interrupt.

Since we do not have timing information, we treat time-triggered tasks as interrupts. For the purpose of this paper, we therefore consider a programming model which consists of a finite set of procedures Proc, a finite collection of interrupt routines Irpt, and a starting node T that serves as an initial task for interrupts to preempt. A procedure or interrupt g is given by a control flow graph (N_g, E_g). Let N denote the union of all sets N_g of program points, extended with T. Likewise, let E denote the union of all sets E_g of control flow edges. Entry and return nodes of g are denoted as s_g and r_g, respectively. Additionally we have a finite set of priorities $\mathbb{P} = \{0, \ldots, p_{max}\}$. In this paper, we assume that each node $u \in N$ is always reached with the same priority $\mathcal{P}(u) \in \mathbb{P}$, where the static initial priority of an interrupt $q \in \mathsf{Irpt}$ is attained at program point s_g and T is the only node with minimal priority, i.e., $\mathcal{P}(T) = 0$. Edges in E are triples (u, cmd, v) leading from program point u to program point v by means of the command cmd. The command cmd either is a basic statement s like an assignment $x = 42$; or a guard $(x \mathrel{!=} y)?$, or a procedure call $h()$ otherwise. For the purpose of this paper we restrict ourselves to global variables only. Therefore, procedures neither need parameters nor return values. The execution of basic statements is assumed to be atomic. How to deal with local variables and parameters is an orthogonal matter and discussed in [6].

In general, the assumption that the dynamic priority is a function of the program point only need not be met by arbitrary OSEK programs. The issue of computing and tracking dynamic priorities, however, is orthogonal to the problem of dealing with flag variables. Therefore, we restrict ourselves to this simple setting, in particular, since it always can be achieved by introducing distinct copies of program points for distinct priorities. For a detailed discussion on computing context and resource aware priorities, see [15].

In order to define data races in a single-core concurrency setting, it is not enough to know that two accesses are both reachable. Whether two access are safe or not is decided by the path the program takes from one access to the other. Therefore, we have chosen a path-based concrete semantics for our model. A similar formalization can be found, e.g., in [8]. Here, a *path* is a sequence of edges where the empty path is denoted by ϵ. For two sets of paths M_1, M_2 the concatenation operator @ appends every path in M_2 to every path in M_1:

$$M_1 @ M_2 = \{\pi_1\pi_2 | \pi_1 \in M_1, \pi_2 \in M_2\}$$

In a first step we characterize the sets of *same-level* paths, i.e., paths leading from the entry node to the return node of the same procedure or interrupt as the least solution of the following constraint system:

$$\begin{array}{lll}
[\mathrm{S0}] & \mathbf{S}[s_g] \supseteq \{\epsilon\} & g \in \mathsf{Irpt} \cup \mathsf{Proc} \\
[\mathrm{S1}] & \mathbf{S}[v] \supseteq \mathbf{S}[u]@\{(u,s,v)\} & (u,s,v) \in E \\
[\mathrm{S2}] & \mathbf{S}[v] \supseteq \mathbf{S}[u]@\mathbf{S}[r_h] & (u,h(),v) \in E \\
[\mathrm{S3}] & \mathbf{S}[u] \supseteq \mathbf{S}[u]@\mathbf{S}[r_q] & q \in \mathsf{Irpt},\ \mathcal{P}(u) < \mathcal{P}(s_q)
\end{array}$$

Given these sets of same-level paths, the sets of paths *reaching* program points from the initial node T are characterized by the following constraint system:

$$\begin{array}{lll}
[\mathrm{R0}] & \mathbf{R}[T] \supseteq \{\epsilon\} & \\
[\mathrm{R1}] & \mathbf{R}[v] \supseteq \mathbf{R}[u]@\{(u,s,v)\} & (u,s,v) \in E \\
[\mathrm{R2}] & \mathbf{R}[v] \supseteq \mathbf{R}[u]@\mathbf{S}[r_h] & (u,h(),v) \in E \\
 & \mathbf{R}[s_h] \supseteq \mathbf{R}[u] & (u,h(),v) \in E \\
[\mathrm{R3}] & \mathbf{R}[s_q] \supseteq \mathbf{R}[u] & q \in \mathsf{Irpt},\ \mathcal{P}(u) < \mathcal{P}(s_q) \\
 & \mathbf{R}[u] \supseteq \mathbf{R}[u]@\mathbf{S}[r_q] & q \in \mathsf{Irpt},\ \mathcal{P}(u) < \mathcal{P}(s_q)
\end{array}$$

In the next step, we introduce a *value* semantics that allows to check whether a path is executable for a given initial state or not. Let D denote the set of program states σ which assign a value to every variable x, e.g. $\sigma\ x = 0$ The concrete value semantics $[\![\cdot]\!]_V : E \to D \dashrightarrow D$ defines for every basic s a *partial* transformer $[\![s]\!] : D \dashrightarrow D$, which we lift to control flow edges (u,s,v) by $[\![(u,s,v)]\!] = [\![s]\!]$. The partial state transformer $[\![\pi]\!]$ for a path $\pi = e_1 \ldots e_n$ then is obtained as the composition of the state transformers $[\![e_i]\!]$ of the edges contained in π, i.e.,

$$[\![\pi]\!] = [\![e_n]\!] \circ \cdots \circ [\![e_1]\!]$$

A path π is *executable* for an initial state σ if $[\![\pi]\!]\ \sigma$ is defined. Let M denote any set of paths. Then the set of paths in M which are executable for σ is given by $\mathit{\Pi}(M,\sigma) = \{\pi \in M \mid \pi \text{ is executable for } \sigma\}$.

Intuitively, a data race occurs at variable x if a low-priority interrupt q_1 accesses x in an edge reaching a program point v_1, at which point it is immediately interrupted by a higher-priority interrupt q and before returning to v_1 the execution reaches another access to x at a program point v_2. Note that, while q_1 is not allowed to continue, q still may itself be interrupted. Therefore v_2 does not necessarily belong to q. Instead, it could belong to another interrupt of even higher priority. This notion is formalized by the following definition.

Definition 1. *There is a data race at variable x with initial state σ if there exists an executable path $\pi \in \Pi(\mathbf{R}[v_2], \sigma)$ reaching some program point v_2 where*

$$\pi = \pi_1(\_, s_1, v_1)(s_q, \_, \_)\pi_2(\_, s_2, v_2)$$

such that the following holds:

(a) s_1 and s_2 are accesses to x;
(b) $\mathcal{P}(v_1) < \mathcal{P}(s_q)$, $\mathcal{P}(v_1) < \mathcal{P}(v_2)$ and also $\mathcal{P}(v_1) < \mathcal{P}(v)$ for every program point v occurring in π_2.

3 Inter-procedural Analysis of Flags

Consider the program in Fig. 1. Race freedom of this program can only be verified by taking the values of the global variable f into account. This variable is set to 1 to flag the critical section of the lower-priority interrupt, and reset to 0 again to signal that the critical section has been left. The information provided by f is respected by the higher-priority interrupt Q in that its value is tested against 0 before Q enters its own critical section. A first practical property of variables f used in such pattern therefore is:

Property 1. f is assigned constants only, f is checked against constants only, and the address of f is never taken.

Also, such synchronization patterns assume an application wide consent about the role of such a variable f and its values. This means that higher-priority interrupts will *respect* the value of f. This leads to our second property:

Property 2. Let p_f denote the least static priority of an interrupt where f is accessed. Then all interrupts of priority exceeding p_f leave f *intact*.

Property 2 means that the value of f before any interrupt q at priority $\mathcal{P}(s_q) > p_f$ equals the value of f after termination of q. Note that we do not rule out that the interrupt q may change the value of f, e.g., in order to protect its own critical section against even higher-priority interrupts as in Fig. 2. In this case, however, q must restore the old value before termination. Subsequently, variables f satisfying Properties 1 and 2 are called *flags*.

3.1 Intra-interrupt Flag Analysis

Let F denote the set of *candidate* flag variables, i.e., all variables which satisfy Property 1. For each $f \in F$ let $\mathbb{V}_f$ denote the set of possible values of f at run-time. Note that F together with all sets $\mathbb{V}_f$ can be easily constructed from the program text. For simplicity assume that for all $f \in F$, $\mathbb{V}_f \subseteq \mathbb{Z}$. Our goal is to compute for each program point u the set of all possible values of all flag variables f when reaching this program point. For this we compute for each flag f and each interrupt or procedure g a summary which maps the value of f before entering g to the set of possible values of f at a program point u in g. This analysis is not new. It is the enhancement of *simple constants* with simple guards [5, 20]. We first require summaries of basic edges. The abstract semantics $[\![\cdot]\!]^\sharp : E \to \mathbb{V}_f \to 2^{\mathbb{V}_f}$ for flag f is given by:

$$\begin{array}{lll} [\![(u, f = c;, v)]\!]^\sharp\ c' = & \{c\} & \\ [\![(u, f \,\Box\, c?, v)]\!]^\sharp\ c' = & \begin{cases} \{c'\} & \text{if } c' \,\Box\, c \\ \emptyset & \text{otherwise} \end{cases} & \\ [\![(u, \_, v)]\!]^\sharp\ c' & = \{c'\} & \text{otherwise} \end{array}$$

Here $\Box$ represents any of the comparison operators $==, !=, <, >$. Due to Property 1, no other assignments or guards involving variable f may occur in the program. First, the effect of a procedure is given by the following constraint system where composition is lifted by $(f \circ g)\ c = \bigcup\{f\ c' \mid c' \in g\ c\}$.

$$\begin{array}{llll} [\mathrm{F}'0] & \mathbf{S}^\sharp[s_h] \sqsupseteq (c \mapsto \{c\}) & & h \in \mathsf{Proc} \\ [\mathrm{F}'1] & \mathbf{S}^\sharp[v] & \sqsupseteq [\![(u, s, v)]\!]^\sharp \circ \mathbf{S}^\sharp[u] & (u, s, v) \in E \\ [\mathrm{F}'2] & \mathbf{S}^\sharp[v] & \sqsupseteq \mathbf{S}^\sharp[r_h] \circ \mathbf{S}^\sharp[u] & (u, h(), v) \in E \end{array}$$

Note that this is the characterization of the flag summary defined according to inter-procedural analysis only, i.e., as if no interrupts may occur. This is made possible due to Property 2 which implies that interrupts, at least when flag variables are concerned, have no impact. Using the inter-procedural summaries, the transformation of flag values from the beginning of the execution of an interrupt to any given program point is characterized by:

$$\begin{array}{llll} [\mathrm{F}0] & \mathbf{R}^\sharp[s_q] \sqsupseteq (c \mapsto \{c\}) & & q \in \mathsf{Irpt} \\ [\mathrm{F}1] & \mathbf{R}^\sharp[v] & \sqsupseteq [\![(u, s, v)]\!]^\sharp \circ \mathbf{R}^\sharp[u] & (u, s, v) \in E \\ [\mathrm{F}2] & \mathbf{R}^\sharp[v] & \sqsupseteq \mathbf{S}^\sharp[r_h] \circ \mathbf{R}^\sharp[u] & (u, h(), v) \in E \\ [\mathrm{F}3] & \mathbf{R}^\sharp[s_h] \sqsupseteq \mathbf{R}^\sharp[u] & & (u, h(), v) \in E \end{array}$$

In practice all flags would be analyzed simultaneously. This analysis is linear in the size of the program and the number of flags. For each flag f it is quadratic in the number of possible values of f. Formally, we show that the analysis is sound — given Property 2 holds.

Lemma 1. *For a program point u, interrupt q, flag f and initial state σ we have for all paths π from s_q to u:*

$$(\llbracket \pi \rrbracket\ \sigma)\ f \in \mathbf{R}^\sharp[u]\ (\sigma\ f)$$

Proof. The proof is by induction on the lengths of paths obtained as solutions of the system [R]. For the initialization constraints [S0], [R0] and [F'0], [F0] we use that $\epsilon \in \Pi(\mathbf{R}[T], \sigma) \subseteq \Pi(\mathbf{R}[s_q], \sigma)$ and with $\llbracket \epsilon \rrbracket\ \sigma = \sigma$ we obtain:

$$(\llbracket \epsilon \rrbracket\ \sigma)\ f\ =\ \sigma\ f\ \in\ \{\sigma\ f\}\ =\ \mathbf{R}^\sharp[s_q]\ (\sigma\ f)$$

Assuming the condition holds for a given path π, we consider how adding an edge (or summary path) and applying the corresponding constraint in the [F/F'] system may change the result. Due to Property 2 interrupts do not contribute to the values of f. Therefore we need not consider constraints [S3] and [R3]. For the extension of π with a basic edge the result follows as a direct consequence of $\llbracket \cdot \rrbracket^\sharp$ soundly abstracting $\llbracket \cdot \rrbracket$. This works for both procedure summaries constructed by [S1] and [F'1] or reaching information constructed by [R1] and [F1]. For same-level paths Constructed by [S2] and [F'2] the result then follows by induction. Consequently we also obtain the result for [R2] and [F2]. Since we know by induction that every program point u that influences a procedure entry node s_h is a sound approximation the result for s_h follows by monotonicity. □

In order to show which programs the analysis can handle precisely we introduce an abstraction of $\llbracket \cdot \rrbracket$ which only evaluates guards on flag f and denote it by $\llbracket \cdot \rrbracket_f$. E.g., $\llbracket (u, x == y, v) \rrbracket_f\ \sigma = \sigma$ for any σ, as the expression $x == y$ does not use the flag f, however $\llbracket (u, f == 5, v) \rrbracket_f\ \sigma = \sigma$ only in the case where $\sigma\ f$ equals 5 — otherwise it is undefined. A path π is f-*executable* for an initial state σ if $\llbracket \pi \rrbracket_f\ \sigma$ is defined. Let M denote any set of paths. Then the set of paths in M which are f-executable for σ is given by $\Pi_f(M, \sigma) = \{\pi \in M \mid \pi \text{ is } f\text{-executable for } \sigma\}$.

Lemma 2. *For a program point u, priority i, flag f and initial state σ we have*

$$\Pi_f(\mathbf{R}[u], \sigma) \neq \emptyset \implies \mathbf{R}^\sharp[u]\ (\sigma\ f) \subseteq \bigcup_{\pi \in \Pi_f(\mathbf{R}[u], \sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

Proof. In case $\mathbf{R}^\sharp[u](\sigma\ f) = \emptyset$ the subset relation holds trivially. Otherwise we do induction on Kleene-iteration steps solving the constraint system [F/F']. First consider the initialization constraints [F0] and [F'0]. For entry nodes s_g we know that $\epsilon \in \Pi_f(\mathbf{R}[s_g], \sigma)$. Since $\llbracket \epsilon \rrbracket$ is the identity function we obtain:

$$\mathbf{R}^\sharp[s_g]\ (\sigma\ f) = \{\sigma\ f\} \subseteq (\llbracket \epsilon \rrbracket\ \sigma)\ f \subseteq \bigcup_{\pi \in \Pi_f(\mathbf{R}[s_g], \sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

For the application of constraints [F1] and [F'1] the result follows since for a flag $\llbracket \cdot \rrbracket^\sharp$ is complete with respect to $\llbracket \cdot \rrbracket$. For constraint [F'2] the result then follows by induction and consequently for [F2]. Regarding [F3] we know by induction, that the condition holds for all program points on the right hand side of the constraint. We have:

$$\mathbf{R}^\sharp[s_h]\ (\sigma\ f) = \bigcup_{(u,h(),v)\in E} (\mathbf{R}^\sharp[u]\ (\sigma\ f)) \subseteq$$
$$\bigcup_{(u,h(),v)\in E}\ \bigcup_{\pi\in\Pi_f(\mathbf{R}[u],\sigma)} \{(\llbracket\pi\rrbracket\ \sigma)\ f\} \subseteq \bigcup_{\pi\in\Pi_f(\mathbf{R}[s_h],\sigma)} \{(\llbracket\pi\rrbracket\ \sigma)\ f\}$$

□

Together Lemmas 1 and 2 show, that if Property 2 holds we lose precision with respect to the concrete semantics only when guards on other variables influence the value of f. Making use of the general properties of OSEK programs we can bootstrap a check of Property 2 from the results of system [F/F'].

Theorem 1. *For a flag f Property 2 holds if for all interrupts q and initial states σ the following holds:*

$$\mathbf{R}^\sharp[r_q]\ (\sigma\ f) = \{\sigma\ f\}$$

Proof. Consider an interrupt q of maximal priority, i.e. $\mathcal{P}(s_q) = p_{max}$. For a program point u occurring in a path $\pi \in \Pi(\mathbf{R}[r_q], \sigma)$ the condition of constraint [S3] is never satisfied since $\mathcal{P}(u) \geq \mathcal{P}(s_q)$. Thus constraint [S3] does not contribute to $\mathbf{R}[r_q]$. Consequently Lemma 1 holds for r_q even when Property 2 is not generally satisfied. Therefore for $\pi \in \Pi(\mathbf{S}[r_q], \sigma)$ and $\mathbf{S}^\sharp[r_q]\ (\sigma\ f) = \{\sigma\ f\}$ we have:

$$\{(\llbracket\pi\rrbracket\ \sigma)\ f\} = \mathbf{S}^\sharp[r_q]\ (\sigma\ f) = \{\sigma\ f\}$$

I.e. Property 2 holds for q. If Property 2 holds for all interrupts of maximal priority we can repeat this argument for the next lower priority and so on. □

4 Simple Analysis of Flags

In this Section we provide a first analysis of flag based synchronization patterns where no value of f is excluded before an interrupt occurs. Accordingly, we start the analysis of each interrupt with $\mathbb{V}_f$. Then by Lemma 1, the set of possible values of f at any program point u is given by

$$\mathbf{R}^\flat[u] = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in \mathbb{V}_f\}$$

Now consider a global variable x. Simple protection mechanisms for x by means of the flag f assume that the value of f after the protected access to x is *reliable.* Here, we call f reliable at a program point u, if its value may not be changed by any higher priority interrupts occurring at u. More formally f is reliable at u if for all interrupts q that may occur at u, i.e. $\mathcal{P}(s_q) > \mathcal{P}(u)$, and every program point v reachable from s_q without entering another interrupt we have:

$$\mathbf{R}^\sharp[v]\ c \subseteq \{c\} \quad \text{for all } c \in \mathbf{R}^\flat[u]$$

This test can be done in time linear in the size of the program and the number of values of f. The following Theorem shows how reliable values of a flag f can be used to verify the absence of data races.

Theorem 2. *Assume that u and v are end points of accesses to x where v is reached without entering another interrupt by an interrupt q with $\mathcal{P}(u) < \mathcal{P}(s_q)$ and flag f is reliable at program point u. Then no data race between u and v occurs, if* $\mathbf{R}^\flat[u] \cap \mathbf{R}^\flat[v] = \emptyset$.

Proof. Assume there is a race path going through u and v. Since the flag f is reliable at u, the sub-path π from u to v may not contain assignments changing f. Let σ be a state reaching u where π is executable for σ and $\sigma\ f = c \in \mathbf{R}^\flat[u]$. Let $\pi = \pi_1(s_q, \_, \_)\pi_2$ where π_2 contains no interrupt entry edges. Then the program state $\sigma_1 = [\![\pi_1]\!]\ \sigma$ after execution of π_1 as well as $[\![\pi_2]\!]\ \sigma_1$ will map f to c. Therefore by Lemma 1, c also must be included in $\mathbf{R}^\flat[v]$. Accordingly, $\mathbf{R}^\flat[u] \cap \mathbf{R}^\flat[v] \neq \emptyset$ — in contradiction to the assumption. $\square$

In the program in Fig. 1, the flag f is reliable at Ix where $\mathbf{R}^\flat[Ix] = \{1\}$ and $\mathbf{R}^\flat[Qx] = \{0\}$. Accordingly due to Theorem 2, the program does not contain a data race. However for the pattern in Fig. 2, the flag is *not* reliable in the sense of our definition here. Therefore, Theorem 2 cannot excluded a data race for variable x.

5 Precise Analysis of Flags

The flag analysis from the last section is imprecise regarding two main points. First, all flag values are assumed to possibly occur at all entry points of interrupts. Second, only *reliable* flag values could be exploited for discarding potential races. In this section we therefore refine our approach by precisely tracking how flag values may change along *inter*-interrupt paths. Due to Property 2 the value analysis from Section 3 still provides valid intra-interrupt results for a given set of flag values at the beginning of the corresponding interrupt. In order to use these results to determine whether a value c is possible for a flag f at the start of an interrupt they are refined by additionally recording the minimal priorities at which f equals c. The minimal priority at which a flag f obtains a certain value is crucial since this is when the value is propagated to the largest number of other interrupts.

Accordingly, we now consider abstract values of the form $\mathbb{V}_f \to \mathbb{P}_\infty$, where $\mathbb{P}_\infty = \mathbb{P} \cup \{\infty\}$ and ∞ denotes, that a value is not obtained at all. The ordering is point-wise and reversed, i.e., $\bot$ maps all values to ∞ (no values are obtained) whereas $\top$ maps all values to 0 (all values can occur at all interrupt entry nodes). The following constraint system [C] accumulates for every interrupt and procedure the occurring values of flag f and the corresponding minimal priorities. In that constraint system, priorities are added to value summaries $\mathbf{R}[u]^\sharp$ of type $\mathbb{V}_f \to 2^{\mathbb{V}_f}$ by means of the functions pry_i which returns a function of type $\mathbb{V}_f \to \mathbb{V}_f \to \mathbb{P}_\infty$ and is defined by:

$$\mathsf{pry}_i\ \phi\ c\ c' = \begin{cases} i & \text{if } c' \in \phi\ c \\ \infty & \text{otherwise} \end{cases}$$

First we characterize functions $\mathbf{S}^*[r_h]$ (h a procedure) where $\mathbf{S}^*[r_h]\ c\ c'$ denotes the minimal priority at which the flag f could take the value c' along any f-executable path in h when the initial value of f is c. For this, we accumulate the values computed by $\mathbf{S}^\sharp[\cdot]$.

$$\begin{array}{llll} [\text{C0}] & \mathbf{S}^*[s_h] \sqsupseteq \mathsf{pry}_{\mathcal{P}(s_h)}\ \mathbf{S}^\sharp[s_h] & & h \in \mathsf{Proc} \\ [\text{C1}] & \mathbf{S}^*[v] \sqsupseteq (\mathsf{pry}_{\mathcal{P}(v)}\ \mathbf{S}^\sharp[v]) \sqcup \mathbf{S}^*[u] & & (u, \_, v) \in E \\ [\text{C2}] & \mathbf{S}^*[v] \sqsupseteq \mathbf{S}^*[r_h] \sqcup \mathbf{S}^*[u] & & (u, h(), v) \in E \end{array}$$

By means of these procedure summaries, we characterize mappings $\mathbf{R}^*[u]$, where $\mathbf{R}^*[u]\ c\ c'$ indicates the minimal priority at which flag f may have value c' along paths reaching u when the value of f equals c at the program point where the last interrupt before reaching u occurs. These are given by the least solution to the following constraint system:

$$\begin{array}{llll} [\text{C3}] & \mathbf{R}^*[s_q] \sqsupseteq \mathsf{pry}_{\mathcal{P}(s_q)}\ \mathbf{R}^\sharp[s_q] & & q \in \mathsf{Irpt} \\ [\text{C4}] & \mathbf{R}^*[v] \sqsupseteq (\mathsf{pry}_{\mathcal{P}(v)}\ \mathbf{R}^\sharp[v]) \sqcup \mathbf{R}^*[u] & & (u, s, v) \in E \\ [\text{C5}] & \mathbf{R}^*[v] \sqsupseteq \mathbf{S}^*[r_h] \sqcup \mathbf{R}^*[u] & & (u, h(), v) \in E \end{array}$$

Since the constraint system [C] is a pure join system (the only operator in right-hand sides is $\sqcup$), the least solution can be computed in time linear in the size of the program and polynomial in the number of flag values. The formal correctness of our construction is given by the following lemma.

Lemma 3. *For a program point u, flag f and initial state σ with $\sigma\ f = c$ the following holds:*

(1) For all paths $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ that do not contain any interrupt edges $(s_q, \_, \_)$ and program points v such that $\pi = \pi_1(\_, \_, v)\pi_2$ and $([\![\pi_1(\_, \_, v)]\!]\ \sigma)\ f = c'$ we have $\mathbf{R}^*[u]\ c\ c' \leq \mathcal{P}(v)$.

(2) If $\mathbf{R}^*[u]\ c\ c' = p$ then there exists a path $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ that does not contain any interrupt edges $(s_q, \_, \_)$ and there exists a program point v such that $\pi = \pi_1(\_, \_, v)\pi_2$ and $([\![\pi_1(\_, \_, v)]\!]\ \sigma)\ f = c'$ and $\mathcal{P}(v) = p$.

Proof. The proof is done by induction on the Kleene-iteration steps used to compute the fix-point. Since we use the same priority information in the systems [C] and [R] Lemmas 1 and 2 imply the result for the initialization constraints [C0] and [C3]. Again by Lemmas 1 and 2 and induction we know the result holds for both operands of a join. Soundness then follows directly. For the completeness statement (2) we note, that $\mathbb{P}_\infty$ is totally ordered. Therefore the join operator element-wise selects the result of one of its operands. □

Consider, e.g., the program in Fig. 1. For the return nodes of r_I and r_Q we have $\mathbf{R}^*[r_I]\ 0 = \{0 \mapsto 1, 1 \mapsto 1\}$ and $\mathbf{R}^*[r_Q]\ 0 = \{0 \mapsto 3, 1 \mapsto \infty\}$, respectively.

The results of the intra-interrupt accumulation of flag values and priorities are now combined to summarize the effects of all interrupts of a given priority. For each priority i, let I_i denote the least upper bound of $\mathbf{R}^*[r_q]$ for all interrupts q of priority i. Thus, $I_i\ c\ c'$ returns the least priority for which the flag f may obtain value c' inside an interrupt of priority i if the value of f is c before execution is interrupted (by an interrupt of priority i). Formally, we have:

Lemma 4. *For a program point u, flag f, initial state σ with $\sigma\ f = c$ and a path $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ with $\pi = (s_q, \_, \_)\pi'$ where π' contains no interrupt edges $(s_{q'}, \_, \_)$ and $([\![\pi]\!]\ \sigma)\ f = c'$, we have $I_{\mathcal{P}(s_q)}\ c\ c' \leq \mathcal{P}(u)$.*

Proof. By definition of I_i we have $I_i\ c\ c' \leq \mathbf{R}^*[r_q]\ c\ c'$ and from Lemma 3 we obtain $\mathbf{R}^*[r_q]\ c\ c' \leq \mathcal{P}(u)$. □

Conversely we can show, that the functions I_i are complete with respect to f-executable paths.

Lemma 5. *For a flag f and initial state σ with $\sigma\ f = c$ if $I_i\ c\ c' = j$ then there exists an interrupt q with $\mathcal{P}(s_q) = i$ and an path $\pi \in \Pi_f(\mathbf{R}[r_q], \sigma)$ such that $\pi = (s_q, \_, \_)\pi_1(\_, \_, v)\pi_2$, π_1 contains no interrupt edges $(s_{q'}, \_, \_)$, $([\![(s_q, \_, \_)\pi_1(\_, \_, v)]\!]\ \sigma)\ f ='\ c$ and $\mathcal{P}(v) = j$.*

Proof. Since $\mathbb{P}_\infty$ is totally ordered, the join on $\mathbb{V}_f \to \mathbb{V}_f \to \mathbb{P}_\infty$ can always select one of the joined values. Therefore we do not introduce finite priorities for values, which are not actually obtained during some interrupt. Let q denote the interrupt, such that $\mathbf{R}^*[r_q]\ c\ c' = j$. Lemma 3 then yields a path realizing the value c. □

In order to account for *inter*-interrupt effects, we first lift the mappings I_i : $\mathbb{V}_f \to \mathbb{V}_f \to \mathbb{P}_\infty$ to mappings $(\mathbb{V}_f \to \mathbb{P}_\infty) \to (\mathbb{V}_f \to \mathbb{P}_\infty)$ by

$$(\mathsf{lift}\ I_i)\ \delta = \delta \sqcup \bigsqcup \{I_i\ c \mid c \in \mathbb{V}_f,\ \delta\ c < i\}$$

Thus, the transformation $(\mathsf{lift}\ I_i)$ takes a mapping of flag values to priorities and returns the mapping, which maps each value c to the minimal priority at which flag f can attain value c if additionally interrupts at priority i are taken into account. Note that the application of $(\mathsf{lift}\ I_i)$ to a mapping δ can be represented as matrix operations and therefore computed in time polynomial in the number of possible values. In the final step, the inter-interrupt summaries $I_{(i,j)}$ comprising the effects of interrupt at priority levels between i and j are obtained from the lifted intra-interrupt summaries by composition:

$$I_{(i,j)} = (\mathsf{lift}\, I_j) \circ \ldots \circ (\mathsf{lift}\, I_i)$$

where for $i > j$, we assume $I_{(i,j)}$ to equal the identity transformation. The level summaries do not need to be composed in every possible order, since a higher priority interrupt must terminate before another interrupt of lower priority can execute. But on termination the higher priority interrupt must have restored the values of flags, i.e. for the lower priority interrupt it might as well not have occurred. Therefore only chains of consecutive interruptions add new entries to the set of possible values of a flag f. Such chains, however, must have strictly increasing priorities. Let $\mathsf{init}_i : 2^{\mathbb{V}_f} \to (\mathbb{V}_f \to \mathbb{P}_\infty)$ denote the functions that given a set of values C returns the mapping which maps each value in C to i, i.e $\mathsf{init}_i\, C\, c = i$ for $c \in C$ and ∞ otherwise. Applying init_0 to the set of initial values of flag f reflects the fact, that initial values can occur at all interrupt entry nodes. Moreover, let $\mathsf{filter}_j : (\mathbb{V}_f \to \mathbb{P}_\infty) \to 2^{\mathbb{V}_f}$ denote the function which takes a mapping of flag values to priorities and returns the set of values of priorities less than j, i.e., $\mathsf{filter}_j\ \delta = \{c \mid \delta\ c < j\}$. Applying filter_j returns the set of flag values possible at entry nodes of interrupts of priority j according to the given mapping. With these definitions, we obtain the set of all possible values of flag f at entry nodes of interrupts with priority j by:

$$C_j = (\mathsf{filter}_j \circ I_{(1,j-1)} \circ \mathsf{init}_0)\ C_0$$

where C_0 is the set of possible values of f at program start. Accordingly, the set of all values of f possibly occurring at program point u when reached by an interrupt with priority j, is given by:

$$\mathbf{R}^\flat[u,j] = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in C_j\}$$

The complexity of applying $I_{(i,j)}$ to some initial mapping of values to priorities, requires to apply $(j-i)$ times a lifted transformation $(\mathsf{lift}\ I_k)$. Therefore, this can be done in time polynomial in the number of possible flag values where the exponent linearly depends on $j-i$.

Theorem 3. *For a program point u reached by an interrupt with priority j, flag f and initial state σ with $\sigma\ f \in C_0$, we have:*

$$\mathbf{R}^\flat[u,j] \supseteq \bigcup_{\pi \in \Pi_f(\mathbf{R}[u],\sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

Proof. For a path $\pi \in \Pi_f(\mathbf{R}[u],\sigma)$ with $(\llbracket \pi \rrbracket\ \sigma)\ f = c''$ we show that $c'' \in \mathbf{R}^\flat[u,j]$. In case that $j = 1$, we have the following:

$$\begin{aligned}\mathbf{R}^\flat[u,j] &= \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in C_1\} = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in (\mathsf{filter}_1 \circ \mathsf{init}_0)\ C_0\} = \\ &\qquad \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in C_0\} \supseteq \mathbf{R}^\sharp[u]\ (\sigma\ f)\end{aligned}$$

With Lemma 1 the result follows. For higher context priorities we decompose π into the two parts. The part inside the final interrupt q and the part before q. Let $\pi = \pi'(\_,\_,v)(s_q,\_,\_)\pi''$ where $\mathcal{P}(s_q) = j$ and π'' does not contain any interrupt edges $(s_{q'},\_,\_)$. Since Lemma 1 applies $(s_{q'},\_,\_)\pi''$ again what's left to show is, that the values observable by $s_{q'}$ are contained in C_j. Let $(\llbracket \pi'(\_,\_,v) \rrbracket\ \sigma)\ f = c'$. Then Lemma 4 yields, that there exists a priority $i \le \mathcal{P}(v)$ such that $I_i\ (\sigma\ f)\ c' = i$. Since $j = \mathcal{P}(s_q) > \mathcal{P}(v)$ we obtain $c' \in C_j$ concluding the proof. □

Note, that Property 2 ensures, that the interrupt-free tails exist. Otherwise higher priority interrupts changing f might be necessary to reach u with c''. For f-executable paths the sets of reaching values are also complete.

Theorem 4. *For a program point u reached by an interrupt with priority j, flag f and initial set C_0 we have:*

$$\mathbf{R}^\flat[u,j] \subseteq \bigcup_{\sigma\ f \in C_0} \ \bigcup_{\pi \in \Pi_f(\mathbf{R}[u],\sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

Proof. In case $\mathbf{R}^\flat[u,j] = \emptyset$ the result follows immediately. Otherwise Lemma 2 yields, that for every $c' \in \mathbf{R}^\flat[u,j]$ there exists a $c'' \in C_j$ such that for a state σ_2 with $\sigma_2\ f = c''$ there exists a path $\pi_2 \in \Pi_f(\mathbf{R}[u],\sigma_2)$ with $(\llbracket \pi_2 \rrbracket\ \sigma_2)\ f = c'$ and $\pi_2 = (s_q, \_, \_)\pi_2'$ with $\mathcal{P}(s_q) = j$. If there exists a program point v with $\mathcal{P}(v) < j$ such that there exists state σ with $\sigma\ f \in C_0$ and a path $\pi_1 in \Pi_f(\mathbf{R}[v],\sigma)$ with $(\llbracket \pi_1 \rrbracket\ \sigma)\ f = c''$ then $\pi_1\pi_2 \in \Pi_f(\mathbf{R}[u],\sigma)$ concluding the proof. If however there is no such state or program point then by Lemma 5 we have $((\mathsf{lift}\ I_{i-1}) \circ \cdots \circ (\mathsf{lift}\ I_1) \circ (\mathsf{init}_0 C_0))\ c'' \geq j$. Therefore filter_j would remove c'' in contradiction with Lemma 2. □

Combining Theorems 3 and 4 shows, that if $\mathbf{R}^\flat[u,j] = \emptyset$ then there is no path which is f-executable for σ_0 and reaches u by an interrupt with priority j.

Corollary 1. *For a program point u, a priority j and a flag f with initial value set $C_0 = \{\sigma_0\ f\}$ we have:*

$$\mathbf{R}^\flat[u,j] = \emptyset \iff \Pi_f(\mathbf{R}[u],\sigma_0) = \emptyset$$

Consider again the program in Fig. 1. For (program points in) the critical sections Ix and Qx we have $\mathbf{R}^\flat[Ix,1]\ \{0\} = \{1\}$ and $\mathbf{R}^\flat[Qx,3]\ \{0\} = \{0\}$.

5.1 Data Race Analysis

In the previous section we have shown, that the values of a flag f can be tracked precisely along all f-executable paths reaching a given program point. A data race, however, consists of two program points u and v at which shared data is accessed and a path from one to the other. Therefore we extend the results for reaching values to start from any given priority level (instead of 1). The set of possible values of flag f at entry nodes of interrupts with priority j from a set of values C at priority i, is given by:

$$C_{i,j} = (\mathsf{filter}_j \circ I_{(i+1,j-1)} \circ \mathsf{init}_i)\ C$$

Accordingly, the set of all values of f possibly occurring at program point u when reached by an interrupt with priority j, is given by:

$$\mathbf{R}^\flat[u,i,j] = \bigcup \{\mathbf{R}^\sharp[u]\ c \mid c \in C_{(i,j)}\}$$

Consider a flag f with initial value set C_0 and program points u and v which are reached by interrupts with priorities j and k, respectively. To decide if there is a race at accesses to a variable x occurring at u and v we first compute by $\mathbf{R}^\flat[u,j] = C$ the values of f reaching u. Then starting with C and the priority $\mathcal{P}(u)$ of u we compute $\mathbf{R}^\flat[v,\mathcal{P}(u),k]$ the set of values of f reaching v is execution is interrupted immediately after the access at u. If this set is non-empty, there exists a path satisfying the conditions for a data race.

In contrast to Theorem 2 the construction and use of interrupt summaries allows us to determine the set of possible values reaching the start of a potentially racing interrupt and avoid the imprecision incurred by starting the analysis with V_f. Furthermore since intermediate changes of flags are tracked it is no longer necessary for flags to be *reliable.* Formally we have:

Theorem 5. *For accesses to a variable x at program points u and v where v is reached by an interrupt with priority $j > \mathcal{P}(u)$ and the set of values of flag f reaching u given by $C \neq \emptyset$ we have:*

(1) There is no race between the two accesses, if $\mathbf{R}^\flat[v,\mathcal{P}(u),j] = \emptyset$.
(2) If $\mathbf{R}^\flat[v,\mathcal{P}(u),j] \neq \emptyset$ there is a race path connecting u and v which is f-executable for some σ_0 with $\sigma_0\ f \in C_0$.

Proof (Theorem 5). If $\mathbf{R}^\flat[v,\mathcal{P}(u),j] = \emptyset$ we have $\mathbf{R}^\sharp[v]\ c = \emptyset$ for all $c \in C_{(\mathcal{P}(u),j)}$. This corresponds to $\mathbf{R}^\flat[v,j] = \emptyset$ in the (sub-)program consisting only of interrupts with priority greater than $\mathcal{P}(u)$. Therefore Corollary 1 yields, that there is no f-executable path reaching v from u with the given value sets. Consequently, there can be no race path.

If however, $\mathbf{R}^\flat[v,\mathcal{P}(u),j] \neq \emptyset$ there exists a path π from u to v which contains only interrupts of priority greater than $\mathcal{P}(u)$ and which is f-executable for initial states σ_1 with $\sigma_1\ f \in C$. Therefore every program point v_1 occurring in π_1 has a higher priority than u, i.e. $\mathcal{P}(v) > \mathcal{P}(u)$. Since $C \neq \emptyset$ there also exists an initial states σ_2 and a path π_2 reaching u that is f-executable for σ_2. With Corollary 1 we can assume, that $(\llbracket\pi_2\rrbracket\ \sigma_2\ f = \sigma_1\ f$. Then $\pi_2\pi_1$ is a f-executable race path since the lowest priority obtained in π_1 is greater than $\mathcal{P}(u)$ and the states match. □

For our running example from Fig. 1 we have the initial value set $\{0\}$ and $\mathbf{R}^\flat[Ix,1] = \{1\}$ as well as $\mathbf{R}^\flat[Qx,1,3] = \emptyset$ since the only value that can pass through the guard edge to Qx is 0. The analysis presented in this section is able to handle more complex patterns as well. For example consider the extension in Fig. 2. The assignment f=2; in the interrupt R adds 2 to the set of values Q can observe at Ix. We have $\mathbf{R}^\flat[I_1,1] = \{1,2\}$. But still $\mathbf{R}^\flat[Qx,1,3] = \emptyset$ certifying, that there is no race between Ix and Qx. If however Q checked for f != 1 instead, which yields the same result for the program without R, we would obtain $\mathbf{R}^\flat[Qx,1,3] = \{2\}$. The corresponding race path is given by first interrupting Ix with R and then after f has been set to 2 interrupting R with Q.

```
int f = 0;              /* Priority 2 */      /* Priority 1 */
int x = 0;              isr_R() {             isr_I () {
/* Priority 3 */          if (f==1){            f = 1;
isr_Q() {                   f = 2;              x++; /* Ix */
  if (f==0)                 f = 1;              f = 0;
    x--; /* Qx */         }                   }
}                       }
```

Fig. 2. Extended example code of program with a flag f

Table 1. Benchmark results

program	loc	time (s)	warnings w/o flags	warnings w flags
`example_flag`	18	0,012	1	0
`resource_flag`	26	0,020	1	0
`inverse_flag`	21	0,012	1	1
`weak_flag`	21	0,016	1	1
`arbiter_flag`	26	0,018	1	0
`linecar`	2586	0,072	5	0
`bipedrobot`	2684	0,028	1	1
`ballsort`	2786	0,424	7	0
`controller`	~400k	3171	929	265

6 Experimental Results

The analysis from Section 4 has been implemented in the Goblint tool [18]. For that, we extended the analysis with tracking of priorities and acquired resources as described in [15], as well as an inter-procedural treatment of local variables and parameters and global invariants to deal approximately with global data different from flag variables. We have tested our implementation on a number of valid and invalid uses of flags. The results are summarized in Table 1. Execution time (in seconds) on an Intel(R) Core(TM) i5 650 running Ubuntu is listed in column 3. The last two columns indicate the numbers of warnings by the analyzer without and with taking flag information into account.

Benchmark `example_flag` is the programs of Fig. 1 and the accesses to x are proven safe. Benchmark `inverse_flag` is benchmark `example_flag` with flipped priorities. While syntactically the flag pattern is intact the associated priorities render it moot. The analysis captures this and issues a data race warning for x. Benchmark `resource_flag` has an additional task of intermediate priority which temporarily (re)sets the flag to 0, but employs resources to increase the priority of I to the intermediate priority before using the flag. Due to the raised priority the value of f is reliable and the analysis verifies the accesses to x as safe. Benchmark `weak_flags` employs the flag pattern correctly, but not thoroughly. There is an additional high priority interrupt, which also accesses x. The analysis recognizes the flag pattern where it is used, but issue a data race warning due to the unprotected access. Benchmark `arbiter_flag` uses an extended synchronization scheme where the lowest priority interrupt uses the value of the flag variable to signal which higher-priority interrupt is allowed to enter its critical section.

Additionally, the analysis has been evaluated on four larger programs Benchmarks `linecar`, `bipedrobot`, and `ballsort` consist of about 300 to 500 lines of application code plus about 2300 lines of headers taken from the NxtOSEK framework [17]. While benchmark `bipedrobot` is taken from the NxtOSEK samples [17], benchmarks `linecar` and `ballsort` have been produced in students' projects. Interestingly, the students made use of flag variables without having been told to do so. Benchmark `linecar` is the control program for a line following car which picks up items placed on the track. Two variables are used to synchronize between scanning for the line and forward movement. Benchmark `bipedrobot` is a two-wheel self balancing robot which uses a resource, i.e. dynamic priorities, to synchronize between balancing and movement. The warning is due to an initialization task, which omits acquiring the resource. It assumes that the timers have not yet triggered the remaining parts of the program. The code of benchmark `ballsort` resembles a state machine controlling a manipulator arm which first locates colored balls in reach and then sorts them by placing red balls to the left and blue balls to the right. Additionally it has a pause button, which stops all movement until it is pressed again. A race-analysis based on priorities alone would warn on all shared variables. On the contrary, treating the state variable as a flag, allows to verify all accesses as safe.

Finally, `controller` is an industrial benchmark. It consists of about 400,000 lines of code including headers. During the analysis 7232 global variables are found. The flag analysis reduces the number of race warnings from 929 to 265. This corresponds to a reduction by over 60%. The analysis on this real-world example takes about 3171s.

Overall, the analysis times, in particular for the larger programs, are acceptable. Also the number of spurious data race warnings is dramatically reduced in the industrial benchmark `controller`. The remaining data race warnings could possibly be reduced further, if additional information such as, e.g., worst-case execution time and cycle duration of time-triggered tasks is taken into account.

7 Related Work

In general terms, we are attempting to exclude invalid paths from diluting the analysis result. Taking into account the control flow of procedure call and return, makes the analysis *context-sensitive*. Static analysis of concurrency errors requires context-sensitive alias analysis, and there are various solutions to obtain context-sensitive alias analysis for race detection, including approaches based on procedure summaries of relative locksets [19], bootstrapping of alias analyses [3], type based label-flow [12], and conditional must-not aliasing [10].

Distinguishing valid paths w.r.t. the sequential semantics of program execution is known as *path-sensitivity*. Path-sensitivity is required to deal, e.g., with conditional locking schemes. Voung et al. [19] identify this as one important source of false alarms in their analyzer. A generic approach to path-sensitivity is *property simulation* [2]. We have previously applied this technique to achieve path-sensitive data race analysis [18]. What we require here, however, is path-sensitivity w.r.t. the interleaving semantics of the program. We need to exclude semantically invalid interleavings, not merely invalid paths in the flow-graphs. None of the static race detection tools we are aware of [3, 9, 13, 19] attempt to exclude semantically invalid interleavings based on boolean guards. Also, our tests with state-of-the-art dynamic race detection tool, Intel Inspector XE 2013 (formerly Intel Thread Checker [14]), revealed that boolean flags confuse the analysis such that no warnings are generated at all even when negating the flag to make it invalid.

We realized the seriousness of this issue when attempted to adapt conventional race detection techniques to priority-driven single-processor concurrency systems [15]. In this setting, locking can be avoided by higher priority threads because lower-priority threads cannot preempt them. We have explored how boolean program variables can be used instead of explicit locking. Another way to avoid acquiring a resources by higher priority threads is to merely check if the lock is free. If no other thread holds the lock, the highest priority thread can simply execute the critical section. Miné [7] presents a scheduled interference semantics of ARINC 653 systems that takes this into account by tracking a separate set of resources known to be free. A mutex is only known to be free when it has been explicitly probed by the highest priority thread that uses it and there has been no blocking operations (e.g., attempting to lock another mutex) since the mutex was last probed.

8 Conclusion

Application developers tend to rely not only on the synchronization mechanisms provided by embedded operating systems such as Autosar/OSEK to secure their programs. Therefore, we provided analysis methods for programs executed by Autosar/OSEK which can deal with hand-crafted synchronization mechanisms based on flag variables.

The analysis is based on an off-the-shelf constant propagation analysis, together with post-processing to take care of the intricate inter-interrupt dependencies arising from the priority based scheduling of OSEK programs. Our characterization of flags allowed us to precisely determine whether two accesses comprise a data race or not. The required fix-point computations are all linear in the size of the program and polynomial in the number of possible flag values, which in our experience is small.

The construction can be enhanced by using more sophisticated value analyses of flag variables, e.g., by allowing to store and restore flag values or by tracking dependences between different flags. Our preliminary experiments, however, showed that already the simple version of our analysis presented in Section 4 drastically reduces the number of false alarms in real-world programs. It remains for future work to evaluate in how far stronger analyses will have a significant impact on the practical precision of the analysis.

Acknowledgements. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement № 269335 (MBAT) and from the German Research Foundation (DFG). The third author was supported by the research theme IUT2-1 and ERDF funded Estonian ICT national programme project “Coinduction”.

References

[1] Autosar consortium: Autosar Architecture Specification, Release 4.0 (2009), `http://www.autosar.org/`

[2] Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI 2002, pp. 57–68. ACM Press (2002)

[3] Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 226–239. Springer, Heidelberg (2007)

[4] 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)

[5] Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206. ACM Press (1973)

[6] Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, pp. 125–140. Springer, Heidelberg (1992)

[7] Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 398–418. Springer, Heidelberg (2011)

[8] Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL 2004, pp. 330–341. ACM Press (2004)

[9] Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI 2006, pp. 308–319. ACM Press (2006)

[10] Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL 2007, pp. 327–338. ACM Press (2007)

[11] OSEK/VDX Group: OSEK/VDX Operating System Specification, Version 2.2.3 (2005), `http://www.osek-vdx.org`

[12] Pratikakis, P., Foster, J.S., Hicks, M.W.: Existential label flow inference via CFL reachability. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 88–106. Springer, Heidelberg (2006)

[13] Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: Context-sensitive correlation analysis for detecting races. In: PLDI 2006, pp. 320–331. ACM Press (2006)

[14] Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: ASID 2006, pp. 34–41. ACM Press (2006)

[15] Schwarz, M.D., Seidl, H., Vojdani, V., Lammich, P., Müller-Olm, M.: Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In: POPL 2011. ACM Press (2011)

[16] Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. 39(9), 1175–1185 (1990)

[17] Chikamasa, T., et al.: OSEK platform for lego® mindstorms® (2010), `http://lejos-osek.sourceforge.net/`

[18] Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155 (2009)

[19] Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: ESEC/FSE 2007, pp. 205–214. ACM Press (2007)

[20] Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 181–210 (1991)

Relational Thread-Modular Static Value Analysis by Abstract Interpretation[⋆]

Antoine Miné

CNRS & École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
mine@di.ens.fr

Abstract. We study thread-modular static analysis by abstract interpretation to infer the values of variables in concurrent programs. We show how to go beyond the state of the art and increase an analysis precision by adding the ability to infer some relational and history-sensitive properties of thread interferences. The fundamental basis of this work is the formalization by abstract interpretation of a rely-guarantee concrete semantics which is thread-modular, constructive, and complete for safety properties. We then show that previous analyses based on non-relational interferences can be retrieved as coarse computable abstractions of this semantics; additionally, we present novel abstraction examples exploiting our ability to reason more precisely about interferences, including domains to infer relational lock invariants and the monotonicity of counters. Our method and domains have been implemented in the AstréeA static analyzer that checks for run-time errors in embedded concurrent C programs, where they enabled a significant reduction of the number of false alarms.

Keywords: static analysis, abstract interpretation, verification, safety, concurrency, embedded programs, rely-guarantee methods.

1 Introduction

Programming is an error-prove activity and software errors are frequent; it is thus useful to design tools that help ensuring program correctness. In this article, we focus on static analyzers, which enjoy several benefits: they are fully automatic (always terminating and requiring minimal annotations, making them easy to deploy and cost-effective in industrial contexts), sound (no program behavior, and so, no bug is overlooked), and they offer a wide range of cost versus precision choices; however they can exhibit false positives (spurious alarms reported by the tool, that need to be checked manually), which we naturally wish to minimize. Abstract interpretation [6] makes it possible to design sound static analyzers in

⋆ This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France and by the project ANR-11-INSE-014 from the French *Agence nationale de la recherche.*

a principled way, by abstraction of a concrete semantics expressing the properties of interest. Prior results on Astrée [3] showed that abstract interpretation could effectively drive the construction of an analyzer that is both efficient and extremely precise (no or few false alarms), by specializing the abstractions to a class of properties and a class of programs, in that case: the absence of run-time error in embedded synchronous control/command avionic C programs. We are now bent on achieving a similar result for *concurrent programs*: we are developing AstréeA [19], a static analyzer to prove the absence of run-time error in embedded concurrent C programs where several threads are scheduled by a real-time operating system, communicate through a shared memory, and synchronize through concurrency primitives (such as mutual exclusion locks).

Although concurrent programming is not new, its use has intensified recently with the rise of consumer multi-core systems. Concurrent programming is also increasingly used to improve cost-effectiveness in the critical embedded domain (e.g., Integrated Modular Avionics [23]), where the need for verification is important. Concurrent programs are more challenging to verify than sequential ones: as a concurrent execution is an interleaving of executions of individual threads often scheduled with a high level of non-determinism (e.g., driven by inputs from the environment), the number of possible executions is generally very high. The verification problem is further complicated by the advent of weakly consistent memories taking hardware and software optimization into account [2].

A solution to avoid considering interleavings explicitly and the associated combinatorial exposition of executions is to use thread-modular methods. Ideally, analyzing a concurrent program should be performed by analyzing individually each thread. Analyzing threads in isolation is not sound as it ignores their potential interactions, but previous work by Carré and Hymans [4] and ourself [19] showed that sequential analyses can be easily modified to take interactions and weakly memory models into account. Unfortunately, these methods are based on a simplistic, non-relational and flow-insensitive concrete semantics of thread interactions, which severely limits the precision of any analysis built by abstracting this semantics. In this article, we propose another, more precise semantics, from which former analyses can be recovered by abstraction, but that also allows more precise, relational abstractions of thread interferences. It is based on Jones' popular rely-guarantee proof method for concurrent programs [13], formulated as a constructive fixpoint semantics in abstract interpretation style.

The rest of this introduction presents our former non-relational interference analysis, exemplifies its shortcommings to motivate our work, and recalls the rely-guarantee reasoning proof technique.

Analysis Based on Non-relational Interferences. We illustrate our former analysis [19] and its limits on the example of Fig. 1. This simple program is composed of two threads: Thread t_2 increments Y by a random value in $[1, 3]$ while it is smaller than 100, and Thread t_1 concurrently increments X while it is smaller than Y. Both variables are initialized to 0 before the program starts.

Consider first the simpler problem of analyzing t_2 in isolation, viewed as a sequential program. We wish to infer the set of reachable memory states

t_1

```
(1a) while random do
       (2a) if $X < Y$ then
              (3a) $X \leftarrow X + 1$
       (4a) endif
(5a) done
```

t_2

```
(1b) while random do
       (2b) if $Y < 100$ then
              (3b) $Y \leftarrow Y + [1,3]$
       (4b) endif
(5b) done
```

Fig. 1. Concurrent program example

$$\begin{array}{ll}
\mathcal{X}_{1b} = \{[X \mapsto 0, Y \mapsto 0]\} & \textit{(initialization)} \\
\mathcal{X}_{2b} = \mathcal{X}_{1b} \cup \mathcal{X}_{5b} & \textit{(control-flow join at loop head)} \\
\mathcal{X}_{3b} = [\![\, Y < 100 \,]\!] \mathcal{X}_{2b} & \textit{(filtering by test condition on } Y\textit{)} \\
\mathcal{X}_{4b} = [\![\, Y \leftarrow Y + [1,3] \,]\!] \mathcal{X}_{3b} & \textit{(assignment into } Y\textit{)} \\
\mathcal{X}_{5b} = \mathcal{X}_{4b} \cup [\![\, Y \geq 100 \,]\!] \mathcal{X}_{2b} & \textit{(control-flow join after test)}
\end{array}$$

Fig. 2. Concrete equation system for Thread t_2 from Fig. 1

(i.e., the values of X and Y) at each program point. This can be expressed classically [7] as the least solution of the invariance equation system in Fig. 2, where each variable $\mathcal{X}_i$ is the memory invariant at program point i, with value in $\mathcal{D} \overset{\text{def}}{=} \mathcal{P}(\{X, Y\} \to \mathbb{Z})$, and $[\![\cdot]\!]$ is the effect of an atomic program operation (assignment or test) on a set of memory states, e.g.: $[\![\, Y < 100 \,]\!]\mathcal{X} \overset{\text{def}}{=} \{\, \rho \in \mathcal{X} \mid \rho(Y) < 100 \,\}$ models a test by filtering states while $[\![\, Y \leftarrow Y + [1,3] \,]\!]\mathcal{X} \overset{\text{def}}{=} \{\, \rho[Y \mapsto \rho(Y) + v] \mid \rho \in \mathcal{X},\, v \in [1,3] \,\}$ models an incrementation by a non-deterministic value. We get, for instance, that the loop invariant at point $2b$ is: $X = 0 \wedge Y \in [0, 102]$. An effective analysis is obtained by replacing concrete variables $\mathcal{X}_i \in \mathcal{D}$ with abstract ones $\mathcal{X}_i^\sharp \in \mathcal{D}^\sharp$ living in an abstract domain $\mathcal{D}^\sharp$, concrete operations $\cup$ and $[\![\cdot]\!]$ with abstract ones $\cup^\sharp$ and $[\![\cdot]\!]^\sharp$, and employing convergence acceleration techniques $\triangledown$ to compute, by iteration, an abstract solution of the system where $\supseteq$ replaces $=$. We get an inductive (but not necessarily minimal) invariant. For Fig. 1, a simple interval analysis using widenings with thresholds can infer that $Y \in [0, 102]$. A similar analysis of t_1 would infer that X and Y stay at 0 as it would ignore, for now, the effect of t_2.

We now turn to the analysis of the full program under the simplest concurrent execution model, sequential consistency [14]: a program execution is an interleaving of tests and assignments from the threads, each operation being considered as atomic. A straightforward approach is to associate a variable $\mathcal{X}_{i,j}$ to each pair of control points, i for t_1 and j for t_2, and construct the product equation system from that of both threads. For instance, we would have:

$$\mathcal{X}_{3a,3b} = [\![\, X < Y \,]\!] \mathcal{X}_{2a,3b} \cup [\![\, Y < 100 \,]\!] \mathcal{X}_{3a,2b} \tag{1}$$

as the point $3a, 3b$ can be reached either with a step by t_1 from $2a, 3b$, or a step by t_2 from $3a, 2b$. However, this quickly results in large equation systems, and we discard this method as impractical. The methods proposed in [4,19] consist instead in analyzing each thread independently as a sequential program, extracting from the analysis (an abstraction of) the set of values each thread stores into each variable, so-called interferences, and then reanalyzing each thread taking into account these interferences; more behaviors of the threads may be exposed in the second run, resulting in more interferences, hence, the thread analyses are iterated until the set of interferences becomes stable. On our example, in the concrete, after the first analysis of t_1 and t_2 reported above, we extract the fact that t_2 can store any value in $[1, 102]$ into Y. In the second analysis of t_1, this information is incorporated by replacing the equation $\mathcal{X}_{3a} = [\![X < Y]\!] \mathcal{X}_{2a}$ with:

$$\mathcal{X}_{3a} = [\![X < (Y \mid [1, 102])]\!] \mathcal{X}_{2a} \tag{2}$$

and similarly for $\mathcal{X}_{5a}$, where $Y \mid [a, b]$ denotes a non-deterministic choice between the current value of Y and an integer between a and b. The test thus reduces to $X < 102$, i.e., t_1 increments X to at most 102. Accordingly, it generates new interferences, on X. As X is not used in t_2, the third analysis round is identical to the second one, and the analysis finishes. By replacing concrete variables, interferences, and operations with abstract ones, and using extrapolation $\triangledown$ to stabilize abstract interferences, we obtain an effective analysis method.

This method is attractive because it is simple and efficient: it is constructed by slightly modifying existing sequential analyses and reuses their abstract domains, it does not require much more memory (only the cost of abstract interferences) nor time (few thread analyses are required in practice, even for large programs, as shown in Fig. 6 in Sec. 5). Unfortunately, modeling interferences as a set of variable values that effect threads in a non-deterministic way severely limits the analysis precision. Even when solving exactly the sequence of concrete equation systems, we can only deduce that $X \in [0, 102] \wedge Y \in [0, 102]$ at the end of Fig. 1 while, in fact, $X \leq Y$ also holds. Naturally, no derived abstract analysis can infer $X \leq Y$, even if it employs a relational domain able to express it (such as octagons [17]).

Relational Rely-Guarantee Reasoning. Rely-guarantee is a proof method introduced by Jones [13] that extends Hoare's logic to concurrent programs. It is powerful enough to prove complex properties, such as $X \leq Y$ in our example. Rely-guarantee replaces Hoare's triples $\{P\}\ s\,\{Q\}$ with quintuples $R, G \vdash \{P\}\ s\,\{Q\}$, requiring us to annotate program points with invariants P and Q, but also relations R and G on whole thread executions; it states that, if the pre-condition P holds before s is executed and all the changes by other threads are included in R, then, after s, Q holds and all the thread's changes are included in G. The annotations required for the program in Fig. 1 are presented in Fig. 3, including the invariants holding at each program point $1a$ to $5b$, and rely assertions R_1, R_2. In particular, to prove that $X \leq Y$ holds in t_1, it is necessary to rely on the fact that t_2 can only increment Y, and so, does not invalidate invariants of the form $X \leq Y$. In Fig. 3, our assertions are very tight, so that each thread exactly guarantees what the other relies on ($R_1 = G_2$ and $R_2 = G_1$). Rely-guarantee is modular: each thread can be checked without looking at the other threads, but only at the rely assertions. This is in contrast to Owicki and checking t_1 : | | checking t_2 : |

t_1	$R_1 = G_2$	$R_2 = G_1$	t_2
$^{(1a)}$ **while** random **do**	X is unchanged	Y is unchanged	$^{(1b)}$ **while** random **do**
$^{(2a)}$ **if** $X < Y$ **then**	Y is incremented	$0 \leq X \leq Y$	$^{(2b)}$ **if** $Y < 100$ **then**
$^{(3a)}$ $X \leftarrow X + 1$	$0 \leq Y \leq 102$		$^{(3b)}$ $Y \leftarrow Y + [1, 3]$
$^{(4a)}$ **endif**			$^{(4b)}$ **endif**
$^{(5a)}$ **done**			$^{(5b)}$ **done**

$1a : X = 0 \wedge Y \in [0, 102]$	$1b : X = 0 \wedge Y = 0$
$2a : X \leq Y \wedge X, Y \in [0, 102]$	$2b : X \leq Y \wedge X, Y \in [0, 102]$
$3a : X < Y \wedge X \in [0, 101] \wedge Y \in [1, 102]$	$3b : X \leq Y \wedge X, Y \in [0, 99]$
$4a : X \leq Y \wedge X, Y \in [1, 102]$	$4b : X \leq Y \wedge X \in [0, 102] \wedge Y \in [1, 102]$
$5a : X \leq Y \wedge X, Y \in [0, 102]$	$5b : X \leq Y \wedge X \in [0, 102] \wedge Y \in [1, 102]$

Fig. 3. Rely-guarantee assertions proving that $X \leq Y$ holds in the program in Fig. 1

Gries' earlier method [21], where checking a Hoare triple required delving into the full code of all other threads to check for non-interference. Intuitively, the rely assertions form an abstraction of the semantics of the threads. While attractive for its expressive power, classic rely-guarantee relies on user annotations. In the following, we use abstract interpretation to infer them automatically.

Overview. The article is organized as follows: Sec. 2 presents the formalization of rely-guarantee in constructive form; Sec. 3 shows how to retrieve our coarse analysis by abstraction while Sec. 4 presents novel abstractions that convey a degree of relationality and history-sensitivity; we also discuss there the analysis in the presence of locks and some uses of trace abstractions. Experimental results are presented in Sec. 5 and Sec. 6 concludes.

Related Work. There is a large body of work on the analysis of concurrent programs; we discuss here only the ones most related to our work and refer the reader to Rinard's survey [22] for general information. We already mentioned previous work on thread-modular static analyses [4,19] which only support non-relational interferences and are limited in precision. Jeannet proposed a precise relational static analysis [12]; it is not thread-modular and may not scale up. Works such as [11] bring thread-modular reasoning to model checking. They inherit the limitations of the underlying model checking method; in the case of [11], the system must be finite-state. Moreover, Malkis et al. observed in [15] that it performs implicitly a non-relational (so-called Cartesian) abstraction; we make here the same observation concerning our previous work [19], but we go further by providing non-trivial relational abstractions. Recent works [10,1] seek to alleviate the burden of providing user annotations in rely-guarantee proof methods, but do not achieve complete automation. Our approach is fundamentally similar to Cousot and Cousot's formulation of the Owicki, Gries, and Lamport proof methods in abstract interpretation form [8], but applied to Jones' method instead. The results in Sec. 2 and Sec. 3 have been partially described before in a research report [20] and course notes [18]; Sec. 4 and Sec. 5 are novel.

2 Rely-Guarantee in Abstract Interpretation Form

The first step in any abstract interpretation is the formalization of the concrete semantics in a constructive form, using fixpoints. We show how, in a very general setting, the concrete semantics of a concurrent program can be presented in a thread-modular way.

2.1 Programs and Transition Systems

Programs. Our programs are composed of a finite set $\mathcal{T}$ of *threads* (the unbounded case is discussed in Sec. 3.3). We denote by $\mathcal{L}$ the set of *program points.* A thread $t \in \mathcal{T}$ is specified as a control-flow graph by: an *entry point* $e_t \in \mathcal{L}$, and a set of *instructions* $\mathit{inst}_t \subseteq \mathcal{L} \times \mathit{Inst} \times \mathcal{L}$. For now, Inst contains assignments $X \leftarrow e$ and comparisons $e \bowtie e'$ (it will be enriched with synchronization primitives in Sec. 4.4). We denote by $\mathcal{V}$ the (possibly unbounded) set of variables; they are global and shared by all the threads. We denote by $\mathbb{V}$ the domain of variable values. To stay general, we deliberately refrain from specifying the set $\mathbb{V}$, the syntax of expressions e, e' and of comparison operators $\bowtie$.

Transition Systems. Following Cousot and Cousot [7], we model program semantics as *labelled transition systems*, a form of small-step semantics which is very general and allows reasoning independently from the chosen programming language. A transition system $(\Sigma, \mathcal{A}, I, \tau)$ is given by: a set Σ of *program states*; a set $\mathcal{A}$ of *actions*; a set $I \subseteq \Sigma$ of *initial states*; a *transition relation* $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$; we will note $\langle \sigma, a, \sigma' \rangle \in \tau$ as $\sigma \xrightarrow{a}_\tau \sigma'$. We instantiate transition systems on our programs as follows:

- $\Sigma \overset{\text{def}}{=} \mathcal{C} \times \mathcal{M}$: states $\langle L, \rho \rangle \in \Sigma$ consist of a *control state* $L \in \mathcal{C} \overset{\text{def}}{=} \mathcal{T} \to \mathcal{L}$ associating a current location $L(t) \in \mathcal{L}$ to each thread $t \in \mathcal{T}$ and a *memory state* $\rho \in \mathcal{M} \overset{\text{def}}{=} \mathcal{V} \to \mathbb{V}$ associating a value $\rho(V) \in \mathbb{V}$ to each variable $V \in \mathcal{V}$;
- $I \overset{\text{def}}{=} \{\, \langle \lambda t.\, e_t,\ \lambda V.\, 0 \rangle \,\}$: we start with all the threads at their entry point and variables at zero;
- $\mathcal{A} \overset{\text{def}}{=} \mathcal{T}$: actions record which thread generates each transition;
- transitions model atomic execution steps of the program:

$$\begin{array}{l}\{\, \langle L, \rho \rangle \xrightarrow{t}_\tau \langle L', \rho' \rangle \mid \langle L(t), \rho \rangle \to_t \langle L'(t), \rho' \rangle \wedge \forall t' \neq t : L(t') = L'(t') \,\} \\ \text{where } \langle \ell, \rho \rangle \to_t \langle \ell', \rho' \rangle \overset{\text{def}}{\iff} \exists i \in \mathit{Inst} : \langle \ell, i, \ell' \rangle \in \mathit{inst}_t \wedge \rho' \in [\![\, i \,]\!] \rho\end{array}$$

  i.e.: we choose a thread t to run and an instruction i from thread t; we let it update t's control state $L(t)$ and the global memory state ρ (the thread transition being denoted as $\langle L(t), \rho \rangle \to_t \langle L'(t), \rho' \rangle$), while the other threads $t' \neq t$ stay at their control location $L(t')$.

2.2 Monolithic Concrete Semantics

We first recall the standard, non-modular definition of the semantics of transition systems. An *execution trace* is a (finite or infinite) sequence of states interspersed with actions, which we denote as: $\sigma_0 \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \cdots$. As we are interested solely in safety properties, our concrete semantics will ultimately compute the so-called *state semantics*, i.e., the set $\mathcal{R}$ of states reachable in any program trace. It is defined classically as the following least fixpoint:[1]

$$\mathcal{R} \stackrel{\text{def}}{=} \operatorname{lfp} R, \text{ where } R \stackrel{\text{def}}{=} \lambda S.\, I \cup \{\, \sigma \mid \exists \sigma' \in S, a \in \mathcal{A} : \sigma' \xrightarrow{a}_{\tau} \sigma \,\} \ . \tag{3}$$

We also recall [5] that $\mathcal{R}$ is actually an abstraction of a more precise semantics: the trace semantics $\mathcal{F}$, that gathers the finite partial traces (i.e., the finite prefixes of the execution traces). The semantics $\mathcal{F}$ can also be defined as a fixpoint:

$$\begin{aligned}&\mathcal{F} \stackrel{\text{def}}{=} \operatorname{lfp} F, \text{ where}\\ &F \stackrel{\text{def}}{=} \lambda X.\, I \cup \{\, \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1} \mid \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \in X \wedge \sigma_i \xrightarrow{a_{i+1}}_{\tau} \sigma_{i+1} \,\} \ .\end{aligned}$$

Indeed, $\mathcal{R} = \alpha^{reach}(\mathcal{F})$, where $\alpha^{reach}(T) \stackrel{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0 \xrightarrow{a_1} \cdots \sigma_n \in T : \exists i \leq n : \sigma = \sigma_i \,\}$ forgets the order of states in traces. The extra precision provided by the trace semantics will prove useful shortly in our thread-modular semantics, and later for history-sensitive abstractions (Sec. 4.3).

The connection with equation systems is well-known: $\mathcal{R}$ is the least solution of the equation $\mathcal{R} = R(\mathcal{R})$. By associating a variable $\mathcal{X}_c$ with value in $\mathcal{P}(\mathcal{M})$ to each $c \in \mathcal{C}$, we can rewrite the equation to the form $\forall c \in \mathcal{C} : \mathcal{X}_c = F_c(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ for some functions F_c. The solution satisfies $\mathcal{R} = \{\, \langle c, \rho \rangle \mid c \in \mathcal{C},\, \rho \in \mathcal{X}_c \,\}$, i.e., $\mathcal{X}_c$ partitions $\mathcal{R}$ by control location. We retrieve standard equation systems for sequential programs (as in Fig. 2) and derive effective abstract static analyses but, when applied to concurrent programs, $\mathcal{C}$ is large and we get unattractively large systems, as exemplified by (1) in the introduction.

2.3 Thread-Modular Concrete Semantics

We can now state our first contribution: a thread-modular expression of $\mathcal{R}$.

Local States. We define the reachable local states $\mathcal{R}l(t)$ of a thread t as the state abstraction $\mathcal{R}$ where the control part is reduced to that of t only. The control part of other threads $t' \neq t$ is not lost, but instead stored in auxiliary variables $pc_{t'}$ (we assume here that $\mathcal{L} \subseteq \mathbb{V}$). Thread local states thus live in $\Sigma_t \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{M}_t$ where $\mathcal{M}_t \stackrel{\text{def}}{=} \mathcal{V}_t \to \mathbb{V}$ and $\mathcal{V}_t \stackrel{\text{def}}{=} \mathcal{V} \cup \{\, pc_{t'} \mid t' \neq t \,\}$. We get:

$$\begin{aligned}&\mathcal{R}l(t) \stackrel{\text{def}}{=} \pi_t(\mathcal{R}) \text{ where}\\ &\pi_t(\langle L, \rho \rangle) \stackrel{\text{def}}{=} \langle L(t), \rho\,[\forall t' \neq t : pc_{t'} \mapsto L(t')] \rangle\\ &\text{extended element-wise as } \pi_t(X) \stackrel{\text{def}}{=} \{\, \pi_t(x) \mid x \in X \,\} \ .\end{aligned} \tag{4}$$

[1] Our functions are monotonic in complete powerset lattices. By Tarski's theorem, all the least fixpoints we use in this article are well defined.

π_t is one-to-one: thanks to the auxiliary variables, no information is lost, which is important for completeness (this will be exemplified in Ex. 3).

Interferences. For each thread $t \in \mathcal{T}$, the interferences it causes $\mathcal{I}(t) \in \mathcal{P}(\Sigma \times \Sigma)$ is the set of transitions produced by t in the partial trace semantics $\mathcal{F}$:

$$\begin{array}{l} \mathcal{I}(t) \stackrel{\text{def}}{=} \alpha^{itf}(\mathcal{F})(t), \text{ where} \\ \alpha^{itf}(X)(t) \stackrel{\text{def}}{=} \{\, \langle \sigma_i,\, \sigma_{i+1} \rangle \mid \exists \sigma_0 \xrightarrow{a_1} \sigma_1 \cdots \xrightarrow{a_n} \sigma_n \in X : a_{i+1} = t \,\} \ . \end{array} \tag{5}$$

Hence, it is a subset of the transition relation τ of the program, reduced to the transitions that appear in actual executions only.

Fixpoint Characterization. $\mathcal{R}l$ and $\mathcal{I}$ can be directly expressed as fixpoints of operators on the transition system, without prior knowledge of $\mathcal{R}$ nor $\mathcal{F}$. We first express $\mathcal{R}l$ in fixpoint form as a function of $\mathcal{I}$:

$$\begin{array}{l} \mathcal{R}l(t) = \operatorname{lfp} R_t(\mathcal{I}), \text{ where} \\ R_t(Y)(X) \stackrel{\text{def}}{=} \pi_t(I) \cup \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X : \sigma \xrightarrow{t}_{\tau} \sigma' \,\} \cup \\ \qquad\qquad\qquad \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X : \exists t' \neq t : \langle \sigma,\, \sigma' \rangle \in Y(t') \,\} \\ R_t \text{ has type: } (\mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma)) \to \mathcal{P}(\Sigma_t) \to \mathcal{P}(\Sigma_t) \ . \end{array} \tag{6}$$

The function $R_t(Y)$ is similar to R used to compute the classic reachability semantics $\mathcal{R}$ in (3) of a thread t, but it explores the reachable states by interleaving two kinds of steps: steps from the transition relation of the thread t, and interference steps from other threads (provided in the argument Y).

Secondly, we express $\mathcal{I}$ in fixpoint form as a function of $\mathcal{R}l$:

$$\begin{array}{l} \mathcal{I}(t) = B(\mathcal{R}l)(t), \text{ where} \\ B(Z)(t) \stackrel{\text{def}}{=} \{\, \langle \sigma,\, \sigma' \rangle \mid \pi_t(\sigma) \in Z(t) \wedge \sigma \xrightarrow{t}_{\tau} \sigma' \,\} \\ B \text{ has type: } (\prod_{t \in \mathcal{T}} \{t\} \to \mathcal{P}(\Sigma_t)) \to \mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma) \ . \end{array} \tag{7}$$

The function $B(Z)(t)$ collects all the transitions in the transition relation of the thread t starting from a local state in $Z(t)$.

There is a mutual dependency between equations (6) and (7), which we solve using a fixpoint. The following theorem, which characterizes reachable local states $\mathcal{R}l$ in a nested fixpoint form, is proved in [18]:

Theorem 1. $\mathcal{R}l = \operatorname{lfp} H$, *where* $H \stackrel{\text{def}}{=} \lambda Z.\, \lambda t.\, \operatorname{lfp} R_t(B(Z))$.

We have the following connection with rely-guarantee proofs $R, G \vdash \{P\}\ s\ \{Q\}$:

– the reachable local states $\mathcal{R}l(t)$ correspond to state assertions P and Q;
– the interferences $\mathcal{I}(t)$ correspond to rely and guarantee assertions R and G;
– proving that a given quintuple is valid amounts to checking that $\forall t \in \mathcal{T}$: $R_t(\mathcal{I})(\mathcal{R}l(t)) \subseteq \mathcal{R}l(t)$ and $B(\mathcal{R}l)(t) \subseteq \mathcal{I}(t)$, i.e., a post-fixpoint check.

Our fixpoints are, however, constructive and can infer the optimal assertions instead of only checking user-provided ones. Computing $\text{lfp}\, R_t(\mathcal{I})$ corresponds to inferring the state assertions P and Q of a thread t given the interferences $\mathcal{I}$, while computing $\text{lfp}\, H$ infers both the interferences and the state assertions.

Thread-modularity is achieved as each function $R_t(Y)$ only explores the transitions $\langle \sigma,\, t,\, \sigma' \rangle$ generated by the thread t in isolation, while relying on its argument Y to know the transitions of the other threads without having to explore them. Note that $R_t(Y)$ has the same control space, $\mathcal{L}$, as the reachability operator R for t considered in isolation. Given an equation system characterizing $\text{lfp}\, R$ after control partitioning: $\forall \ell \in \mathcal{L} : \mathcal{X}_\ell = F_\ell(\mathcal{X}_1, \ldots, \mathcal{X}_n)$, $\text{lfp}\, R_t(Y)$ can be characterized very similarly, as $\forall \ell \in \mathcal{L} : \mathcal{X}_\ell = F'_\ell(\mathcal{X}_1, \ldots, \mathcal{X}_n) \cup \mathit{apply}_\ell(Y)(\mathcal{X}_\ell)$, where each F'_ℓ extends F_ℓ to pass auxiliary variables unchanged, but is still defined only from the instructions in inst_t, while apply_ℓ applies interferences from Y at ℓ. Hence, Y being fixed, $R_t(Y)$ is similar to an analysis in isolation of t. Finally, computing $\text{lfp}\, H$ by iteration corresponds to reanalyzing threads with $R_t(Y)$ until Y stabilizes. Although the semantics is concrete and uncomputable, we already retrieve the structure of the thread-modular static analysis from previous work [19] recalled in Sec. 1.

Completeness. Given any $\mathcal{R}l(t)$, we can easily recover $\mathcal{R}$ as $\mathcal{R} = \pi_t^{-1}(\mathcal{R}l(t))$ because π_t is one-to-one. We deduce that Thm. 1 gives a complete method to infer all safety properties of programs.

Example 1. Consider our example from Fig. 1. We do not present $\mathcal{R}l$ and $\mathcal{I}$ in full as these are quite large; we focus on the interferences generated by t_2 at point $3b$. They have the form $\langle\langle(\ell, 3b),\, (x, y)\rangle,\, \langle(\ell, 4b),\, (x, y')\rangle\rangle$ where $y \in [0, 99]$, $y' \in [y+1, y+3]$, and $x = 0$ if $\ell = 1a$, $x \in [0, y]$ if $\ell = 2a$ or $\ell = 5a$, $x \in [0, y-1]$ if $\ell = 3a$, and $x \in [1, y]$ if $\ell = 4a$. Note that, in the full transition relation τ, $Y \leftarrow Y + [1, 3]$ generates a much larger set of transitions: $\langle\langle(\ell, 3b),\, (x, y)\rangle,\, t_2,\, \langle(\ell, 4b),\, (x, y')\rangle\rangle$ where $y' \in [y+1, y+3]$, with no constraint on x nor y.

3 Retrieving Existing Analyses

We now express our former analysis based on non-relational and flow-insensitive interferences as an abstraction of the thread-modular concrete semantics.

3.1 Flow-Insensitive Abstraction

A first abstraction consists in reducing the domains by forgetting as much control information as possible. In order to avoid losing too much precision, individual thread analyses should remain flow-sensitive with respect to their own control location. Thus, on local states, we remove the auxiliary variables using an abstraction $\alpha_{\mathcal{R}}^{nf}$ from $\mathcal{P}(\mathcal{L} \times \mathcal{M}_t)$ to $\mathcal{P}(\mathcal{L} \times \mathcal{M})$ and, on interferences, we remove the control part entirely using an abstraction $\alpha_{\mathcal{I}}^{nf}$ from $\mathcal{P}(\Sigma \times \Sigma)$ to $\mathcal{P}(\mathcal{M} \times \mathcal{M})$:

$$\begin{aligned} \alpha_{\mathcal{R}}^{nf}(X) &\stackrel{\text{def}}{=} \{\, \langle \ell,\, \rho_{|\mathcal{V}} \rangle \mid \langle \ell,\, \rho \rangle \in X \,\} \\ \alpha_{\mathcal{I}}^{nf}(Y) &\stackrel{\text{def}}{=} \{\, \langle \rho,\, \rho' \rangle \mid \exists L, L' \in \mathcal{C} :\, \langle\langle L,\, \rho\rangle,\, \langle L',\, \rho'\rangle\rangle \in Y \,\} \ . \end{aligned}$$

Applying these abstractions to R_t and B gives rise to the following coarser version of (6)–(7), from which we derive an approximate fixpoint semantics $\mathcal{RI}^{nf}$:

$$\begin{array}{l}
\mathcal{RI}^{nf} \stackrel{\text{def}}{=} \operatorname{lfp} \lambda Z.\, \lambda t.\, \operatorname{lfp} R_t^{nf}(B^{nf}(Z)), \text{ where} \\
B^{nf}(Z)(t) \stackrel{\text{def}}{=} \{\, \langle \rho,\, \rho' \rangle \mid \exists \ell, \ell' \in \mathcal{L} :\, \langle \ell,\, \rho \rangle \in Z(t) \wedge \langle \ell,\, \rho \rangle \to_t \langle \ell',\, \rho' \rangle \,\} \\
R_t^{nf}(Y)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup A_t^{nf}(Y)(X) \\
R_t^{loc}(X) \stackrel{\text{def}}{=} \{\langle e_t,\, \lambda V.\, 0 \rangle\} \cup \{\, \langle \ell',\, \rho' \rangle \mid \exists \langle \ell,\, \rho \rangle \in X :\, \langle \ell,\, \rho \rangle \to_t \langle \ell',\, \rho' \rangle \,\} \\
A_t^{nf}(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell,\, \rho' \rangle \mid \exists \rho, t' \neq t :\, \langle \ell,\, \rho \rangle \in X \wedge \langle \rho,\, \rho' \rangle \in Y(t') \,\} \ .
\end{array} \tag{8}$$

We retrieve in R_t^{nf} the interleaving of local transitions R_t^{loc} and interferences $A_t^{nf}(Y)$ from Y. Interferences are handled in a flow-insensitive way: if a thread t' can generate a transition between two memory states at some point, we assume that it can happen at any point in the execution of t. $\mathcal{RI}^{nf}$ could be turned into an effective static analysis by reusing stock abstractions for memory states $\mathcal{P}(\mathcal{M})$ and relations $\mathcal{P}(\mathcal{M} \times \mathcal{M})$; however, abstracting relations can be inefficient in the large and we will abstract interferences further in the next section.

Example 2. When computing $\mathcal{RI}^{nf}$ in Fig. 1, we obtain the assertions in Fig. 2. For instance, the interferences from t_2 are $\{\, \langle (x,y),\, (x',y') \rangle \mid x = x', y \leq y' \leq y+3, x, y \in [0,99], x \leq y \,\}$. This shows that auxiliary variables and flow-sensitive interferences are not always necessary to infer interesting properties.

Example 3. Consider a program composed of two identical threads reduced to an incrementation: t_1 is ${}^{(1a)}X \leftarrow X + 1^{(2a)}$ and t_2 is ${}^{(1b)}X \leftarrow X + 1^{(2b)}$. At $2a$, the state with auxiliary variables is $(pc_2 = 1b \wedge X = 1) \vee (pc_2 = 2b \wedge X = 2)$. It implies $X \in [1,2]$, but also the fact that t_2 can only increment X when $pc_2 = 1b$, i.e., when $X = 1$. If we forget the auxiliary variables, we also forget the relation between pc_2 and X, and no upper bound on X is stable by the effect of t_2; we get the coarser invariant: $X \geq 1$. We retrieve here a classic result: modular reasoning on concurrent programs is not complete without auxiliary variables.

3.2 Non-relational Interference Abstraction

After removing control information, interferences live in $\mathcal{P}(\mathcal{M} \times \mathcal{M})$. Such relations provide two flavors of relationality: input-output relationality and relationships between variable values. To recover the analysis described in Sec. 1, we only remember which variables are modified by interferences and their new value, using the following abstraction $\alpha_{\mathcal{I}}^{nr}$ from $\mathcal{P}(\mathcal{M} \times \mathcal{M})$ to $\mathcal{V} \to \mathcal{P}(\mathbb{V})$:

$$\alpha_{\mathcal{I}}^{nr}(Y) \stackrel{\text{def}}{=} \lambda V.\, \{\, x \in \mathbb{V} \mid \exists \langle \rho,\, \rho' \rangle \in Y :\, \rho(V) \neq x \wedge \rho'(V) = x \,\} \tag{9}$$

which forgets variable relationships as it abstracts each variable separately, and all but the simplest input sensitivity. Applying this abstraction to the flow-insensitive interference semantics (8), we derive the following, further approximated fixpoint semantics:

$$\begin{aligned}
&\mathcal{Rl}^{nr} \stackrel{\text{def}}{=} \operatorname{lfp} \lambda Z.\, \lambda t.\, \operatorname{lfp} R_t^{nr}(B^{nr}(Z)), \text{ where} \\
&B^{nr}(Z)(t) \stackrel{\text{def}}{=} \alpha_{\mathcal{I}}^{nr}(B^{nf}(Z)(t)) \\
&R_t^{nr}(Y)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup A_t^{nr}(Y)(X) \\
&A_t^{nr}(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell,\, \rho[V \mapsto v] \rangle \mid \langle \ell,\, \rho \rangle \in X,\, V \in \mathcal{V},\, \exists t' \neq t : v \in Y(t')(V) \,\} \enspace .
\end{aligned} \tag{10}$$

Example 4. When computing $\mathcal{Rl}^{nr}$ in Fig. 1, we obtain the abstract interferences $[X \mapsto [1, 102], Y \mapsto \emptyset]$ for t_1 and $[X \mapsto \emptyset, Y \mapsto [1, 102]]$ for t_2, which is sufficient to infer precise bounds for X and Y, but not to infer the relation $X \leq Y$: when t_1 is analyzed, we allow t_2 to store any value from $[1, 102]$ into Y, possibly decrementing Y and invalidating the relation $X \leq Y$.

Soundness. The soundness of (8) and (10) is stated respectively as $\forall t \in \mathcal{T}$: $\mathcal{Rl}^{nf}(t) \supseteq \alpha_{\mathcal{R}}^{nf}(\mathcal{Rl}(t))$ and $\mathcal{Rl}^{nr}(t) \supseteq \alpha_{\mathcal{R}}^{nr}(\alpha_{\mathcal{R}}^{nf}(\mathcal{Rl}(t)))$. It is a consequence of the general property: $\alpha(\operatorname{lfp} F) \subseteq \operatorname{lfp} F^\sharp$ when $\alpha \circ F \subseteq F^\sharp \circ \alpha$ [5, Thm. 1]. This soundness proof is far simpler than the ad-hoc proof from [19], and we find it more satisfying to construct systematically a sound analysis by abstraction of a concrete semantics rather than presenting an analysis first and proving its soundness *a posteriori.*

Static Analysis. We can construct a static analysis based on (10): state sets X are abstracted by associating to each program point an element of a (possibly relational) domain abstracting $\mathcal{P}(\mathcal{M})$; interferences Y associate to each thread and variable in $\mathcal{V}$ an abstract value abstracting $\mathcal{P}(\mathbb{V})$ (for instance, an interval).

Actually, partitioning R_t^{nr} does not give the equations in Sec. 1 and [19], but an alternate form where interferences are applied on all variables at all equations. For instance, instead of $\mathcal{X}_{3a} = [\![\, X < (Y \mid [1, 102]) \,]\!] \mathcal{X}_{2a}$ (2), we would get:

$$\mathcal{X}'_{3a} = [\![\, X < Y \,]\!] \mathcal{X}'_{2a} \cup [\![\, Y \leftarrow [1, 102] \,]\!] \mathcal{X}'_{3a} \enspace .$$

The first form (2) is more efficient as it takes interferences into account lazily, when reading variables, and it avoids creating extra dependencies in equations. The correctness of this important optimization is justified by the fact that the variables $\mathcal{X}_\ell$ in (2) actually represent local states up to pending interferences. Given the non-relational interferences $Y \in \mathcal{T} \to \mathcal{V} \to \mathcal{P}(\mathbb{V})$, we have: $\mathcal{X}'_\ell = \{\, \rho \mid \exists \rho' \in \mathcal{X}_\ell : \forall V : \rho(V) = \rho'(V) \vee \exists t' \neq t : \rho(V) \in Y(t')(V) \,\}$. The operators $[\![\cdot]\!]$ are modified accordingly to operate on pairs $\langle \mathcal{X}_\ell,\, Y \rangle$ instead of $\mathcal{X}'_\ell$, as shown in (2) and, more systematically, in [19].

3.3 Unbounded Thread Instances

Up to now, we have assumed that the set $\mathcal{T}$ of threads is finite. Allowing an infinite $\mathcal{T}$ is useful, however, to analyze programs with an unbounded number of threads. We consider, in this section only, the useful case where $\mathcal{T}$ is composed of a finite set $\mathcal{T}_s$ of syntactic threads, a subset of which $\mathcal{T}_\infty \subseteq \mathcal{T}_s$ can appear more than once (and possibly infinitely often) in $\mathcal{T}$.

The fixpoint formulations of Thm. 1, as well as (8), (10), (11) do not require a finite $\mathcal{T}$; an infinite $\mathcal{T}$ still results in a well defined, if uncomputable, concrete semantics. Finiteness is required to construct an effective static analysis, for three reasons: (*i*) iterating over the threads in Thm. 1 should terminate, (*ii*) control states must be finitely representable, and (*iii*) maps from threads to abstract interferences must be finitely representable. Applying the flow-insensitive abstraction from Sec. 3.1 removes infinite control states, solving (*ii*). As the local states and interferences of two instances of a syntactic thread are then isomorphic, we abstract threads from $\mathcal{T}$ to $\mathcal{T}_s$ by storing information for and iterating over only one instance of each thread in $\mathcal{T}_\infty$, solving (*i*) and (*iii*). This abstraction changes slightly the interpretation of the test $t' \neq t$ when applying interferences. For instance, A_t^{nr} from (10) is changed into:

$$A_t^{nr}(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell, \rho[V \mapsto v] \rangle \mid \langle \ell, \rho \rangle \in X \wedge \exists t' : (t \neq t' \vee t \in \mathcal{T}_\infty) \wedge v \in Y(t')(V) \,\}$$

i.e., we consider self-interferences for threads with several instances. This abstraction makes the analysis of programs with an unbounded number of threads possible, but with some limit on the precision. The resulting analysis is uniform: it cannot distinguish between different instances of the same thread nor express properties that depend on the actual number of running threads.

4 Relational Interferences

We now construct novel interference abstractions that enjoy a level of relationality and flow-sensitivity. We apply them on some examples, including Fig. 1.

4.1 Invariant Interferences

The non-relational abstraction of Sec. 3.2 applies interferences independently to each variable, destroying any relationship. To improve the precision, we infer relationships maintained by interferences, i.e., holding both before and after the interference. We use the following abstraction $\alpha_{\mathcal{I}}^{inv}$ from $\mathcal{P}(\mathcal{M} \times \mathcal{M})$ to $\mathcal{P}(\mathcal{M})$, which joins the domain and the codomain of a relation on states:

$$\alpha_{\mathcal{I}}^{inv}(Y) \stackrel{\text{def}}{=} \{\, \rho \mid \exists \rho' : \langle \rho, \rho' \rangle \in Y \vee \langle \rho', \rho \rangle \in Y \,\} \ .$$

Note that this abstraction is able to express relations between variables modified by a thread and variables not modified, such as $X \leq Y$ for t_2 in Fig. 1. However, unlike our former abstraction $\alpha_{\mathcal{I}}^{nr}$ (10), $\alpha_{\mathcal{I}}^{inv}$ forgets which variables have been modified. To construct our analysis, we thus combine them in a *reduced product*:

$$\begin{array}{l}
\mathcal{R}l^{rel} \stackrel{\text{def}}{=} \text{lfp}\, \lambda Z.\, \lambda t.\ \text{lfp}\, R_t^{rel}(B^{rel}(Z)), \text{ where} \\
B^{rel}(Z) \stackrel{\text{def}}{=} \langle \lambda t.\, \alpha_{\mathcal{I}}^{nr}(B^{nf}(Z)(t)),\ \lambda t.\, \alpha_{\mathcal{I}}^{inv}(B^{nf}(Z)(t)) \rangle \\
R_t^{rel}(\langle Y^{nr}, Y^{inv} \rangle)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup (A_t^{nr}(Y^{nr})(X) \cap A_t^{inv}(Y^{inv})) \\
A_t^{inv}(Y^{inv}) \stackrel{\text{def}}{=} \{\, \langle \ell, \rho \rangle \mid \ell \in \mathcal{L},\ \rho \in Y^{inv}(t) \,\} \ .
\end{array} \qquad (11)$$

Designing a static analysis derived on this abstraction is straightforward. Interference invariants $Y^{inv}(t) \in \mathcal{P}(\mathcal{M})$ are abstracted in any classic domain (e.g., octagons [17] to express $X \le Y$). Computing abstract invariants $\alpha_{\mathcal{I}}^{inv}(B^{nf}(Z)(t))$ reduces to computing the abstract join $\cup^\sharp$ of the abstract environments of all the program points of t. Applying abstract interferences in R_t^{rel} reduces to standard abstract set operators $\cup^\sharp$ and $\cap^\sharp$. A drawback is that the optimization used in (2) to apply interferences lazily, only at variable reads, can no longer be performed here, resulting in a much slower analysis. We will alleviate the problem in Sec. 4.4 by using relational invariant interferences only at a few key locations.

4.2 Monotonicity Interference

We now give an example abstraction providing input-output relationality on interferences. In order to complete the analysis of Fig. 1, we propose a simple domain that infers the monotonicity of variables. Interferences are abstracted from $\mathcal{P}(\mathcal{M} \times \mathcal{M})$ to maps $\mathcal{V} \to \mathbb{D}$, where $\mathbb{D} \overset{\text{def}}{=} \{\nearrow, \top\}$ indicates whether each variable is monotonic ($\nearrow$) or not ($\top$), hence the following abstraction:

$$\alpha_{\mathcal{I}}^{mon}(Y) \overset{\text{def}}{=} \lambda V.\,\text{if } \forall \langle \rho,\, \rho' \rangle \in Y : \rho(V) \le \rho'(V) \text{ then } \nearrow \text{ else } \top \ .$$

We would, as before, apply $\alpha_{\mathcal{I}}^{mon}$ to (8) and combine it with (10) or (11) to get a new reduced product fixpoint semantics. We do not present these formulas, but rather focus on the key operations in a static analysis. Firstly, we infer approximate monotonicity information for interferences $\alpha_{\mathcal{I}}^{mon}(B^{nf}(Z)(t))$: during the analysis of t, we gather, for each variable V, the set of all the assignments into V, and set V to $\nearrow$ if they all have the form $V \leftarrow V + e$ where e evaluates to positive values, and set V to $\top$ otherwise. Secondly, we use monotonicity information when applying interferences after an affine comparison operator $e_1 \le e_2$: if all the variables in e_1 and e_2 have monotonic interferences and appear in e_2 (resp. e_1) with positive (resp. negative) coefficient, then $[\![\, e_1 \le e_2 \,]\!]$ can be applied *after* applying the non-relational interferences. In Fig. 1, for instance, we would get: $\mathcal{X}_{3a} = [\![\, X < Y \,]\!]([\![\, X < (Y|[1,102]) \,]\!]\mathcal{X}_{2a})$ instead of (2). Using these abstractions, we can prove that, at the end of the program, $X \le Y$ holds. The domain is inexpensive as it associates a single binary information to each variable.

4.3 Trace Abstractions

Although state semantics are complete for safety properties, it is often useful to start from a more expressive, trace concrete semantics to embed some information about the history of computations in subsequent abstractions. A classic example is trace partitioning [16] where, in order to avoid or delay abstract joins (which often cause imprecision), a disjunction of abstract elements is maintained, each one keyed to an abstraction of sequences of control locations leading to the current location (such as, which branch was followed in the previous test).

We can construct a trace version of the thread-modular concrete semantics from Sec. 2.3 and its abstractions from Sec. 3 and Sec. 4 by upgrading R_t to

t_1

```
**while** random **do**
  **if** $H < 10,000$ **then**
    $H \leftarrow H + 1$
  **endif**
**done**
```

t_2

```
**while** random **do**
  $C \leftarrow H$
**done**
```

t_3

```
**while** random **do**
  **if** random **then** $T \leftarrow 0$
  **else** $T \leftarrow T + (C - L)$ **endif**
  $L \leftarrow C$
**done**
```

Fig. 4. Clock example motivating history-sensitive invariants

append states to partial executions. Applying this idea, for instance, to the flow-insensitive semantics of Sec. 3.1 which is the basis of our abstractions, we get:

$$\begin{array}{l}
R_t^{nf}(Y)(X) \overset{\text{def}}{=} \{\langle e_t, \lambda V.0\rangle\} \cup \\
\{\, \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle, \langle \ell', \rho'\rangle\rangle \mid \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle\rangle \in X, \langle \ell, \rho\rangle \rightarrow_t \langle \ell', \rho'\rangle \,\} \cup \\
\{\, \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle, \langle \ell, \rho'\rangle\rangle \mid \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle\rangle \in X, \exists t' \neq t : \langle \rho, \rho'\rangle \in Y(t') \,\} \\
R_t^{nf} \text{ has type: } (\mathcal{T} \rightarrow \mathcal{P}(\mathcal{M} \times \mathcal{M})) \rightarrow \mathcal{P}((\mathcal{L} \times \mathcal{M})^*) \rightarrow \mathcal{P}((\mathcal{L} \times \mathcal{M})^*)
\end{array}$$

From the point of view of thread t, an execution is composed of a sequence of local states (without auxiliary variables) in $\mathcal{L} \times \mathcal{M}$; it starts at its entry point e_t and is extended by either an execution step $\langle \ell, \rho\rangle \rightarrow_t \langle \ell', \rho'\rangle$ of thread t or an interference form Y that leaves its local control location unchanged. The semantics can be translated, as before, into an equation system resembling that of the sequential analysis of the thread t in isolation (Fig. 2) by associating to each control location $\ell \in \mathcal{L}$ a variable $\mathcal{X}_\ell$ that stores (an abstraction of) the partial traces that end in the control state ℓ. A natural consequence is the ability to use classic trace partitioning techniques intended for sequential programs [16] when analyzing each thread, independently from interferences.

We illustrate the use for concurrency-specific trace abstractions on the example in Fig. 4. This program, inspired from an actual software, contains three threads: t_1 increments a clock H, t_2 samples the clock in a latch C, and t_3 accumulates elapsed durations with respect to C into T. We wish to infer that $T \leq L \leq C \leq H$, i.e., the accumulated time does not exceed the total elapsed time. This information can be inferred from the monotonicity of L, C, and H; for instance, the assignment $L \leftarrow C$ where C is monotonic implies that $L \leq C$ holds despite interferences. However, the monotonicity domain of Sec. 4.2 can only infer the monotonicity of H, not that of C. In particular, in that domain, it would be unsound for the semantics $[\![\, C \leftarrow H \,]\!]^\sharp \mathcal{X}^\sharp$ to deduce the monotonicity of C from that of H. Otherwise, in the following example, if both H and H' were monotonic, we would deduce wrongly that C is also monotonic:

$$\textbf{if } \text{random } \textbf{then } C \leftarrow H \textbf{ else } C \leftarrow H' \textbf{ endif} \ . \tag{12}$$

We need to infer a stronger property, namely that the sequence of values stored into C is a subsequence of the values stored into H. This is implied by the assignment $C \leftarrow H$ but not by (12), and it implies the monotonicity of C. The subsequence abstraction $\alpha_{\mathcal{R}}^{sub}$ is an abstraction from sequences of states local to a thread, i.e., in $\mathcal{P}((\mathcal{L} \times \mathcal{M})^*)$, to $\mathcal{V} \rightarrow \mathcal{P}(\mathcal{V})$, which is defined as:

$$\begin{aligned}\alpha^{sub}_{\mathcal{R}}(X)(V) \stackrel{\text{def}}{=} \{\, W \mid \forall \langle \langle \ell_0,\, \rho_0\rangle, \ldots, \langle \ell_n,\, \rho_n\rangle\rangle \in X : \exists i_0, \ldots, i_n : \\ \forall k : i_k \le k \wedge i_k \le i_{k+1} \wedge \forall j : \rho_j(V) = \rho_{i_j}(W) \,\} \ .\end{aligned}$$

It associates to each variable V the set of variables W it can be considered a subsequence of. Some meaningful subsequence information can be inferred by only looking at simple assignments of the form $V \leftarrow W$. The domain is inexpensive; yet, in a reduced product with the monotonicity domain, it allows inferring all the properties required to precisely analyze Fig. 4.

4.4 Lock Invariants

We now enrich our programs with mutual exclusion locks, so-called *mutexes*, to provide thread synchronization. We assume a finite set $\mathbb{M}$ of mutexes and two instructions: **lock**(m) and **unlock**(m), to respectively acquire and release a mutex $m \in \mathbb{M}$. The semantics is that, at any time, each mutex can be held by at most one thread. To reflect this, our transition systems are enriched as follows:

– $\Sigma \stackrel{\text{def}}{=} \mathcal{C} \times \mathcal{M} \times \mathcal{S}$: states $\langle L,\, \rho,\, s\rangle \in \Sigma$ include a new *scheduler component* $s \in \mathcal{S} \stackrel{\text{def}}{=} \mathbb{M} \to (\mathcal{T} \cup \{\bot\})$ which remembers, for each mutex, which thread holds it, if any, or $\bot$ if the mutex is unlocked;
– $I \stackrel{\text{def}}{=} \{\langle \lambda t.\, e_t,\, \lambda V.\, 0,\, \lambda m.\, \bot\rangle\}$: all mutexes are initially unlocked;
– instructions $\langle \ell,\, \mathbf{lock}(m),\, \ell'\rangle \in inst_t$ and $\langle \ell,\, \mathbf{unlock}(m),\, \ell'\rangle \in inst_t$ generate respectively the following transition sets:

$$\begin{aligned}&\{\, \langle L[t \mapsto \ell],\, \rho,\, s\rangle \xrightarrow{t}_{\tau} \langle L[t \mapsto \ell'],\, \rho,\, s[m \mapsto t]\rangle \mid \langle L,\, \rho,\, s\rangle \in \Sigma,\, s(m) = \bot \,\} \\ &\{\, \langle L[t \mapsto \ell],\, \rho,\, s\rangle \xrightarrow{t}_{\tau} \langle L[t \mapsto \ell'],\, \rho,\, s[m \mapsto \bot]\rangle \mid \langle L,\, \rho,\, s\rangle \in \Sigma,\, s(m) = t \,\} \ .\end{aligned}$$

Consider the example in Fig. 5.(a), where two identical threads increment a counter X up to 100. The use of a mutex m ensures that X is not modified between the test $X < 100$ and the subsequent incrementation. Ignoring the mutex in the concrete would give a range of $[0, 101]$ instead of $[0, 100]$ and, with flow-insensitive interferences, we would not find any upper bound on X (the case is similar to Ex. 3). We now show that partitioning with respect to the scheduler state (a technique introduced in [19]) can make the analysis more precise.

Returning to our most concrete, thread-modular semantics of Sec. 2.3, we enrich the local states $\mathcal{R}l$ of a thread t with information on the set of locks it holds: $\mathcal{R}l(t) \in \Sigma_t \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{M}_t \times \mathcal{P}(\mathbb{M})$. We define $\mathcal{R}l(t) \stackrel{\text{def}}{=} \pi_t(\mathcal{R})$ where the projection π_t to local states from (4) is extended to handle $s \in \mathcal{S}$ as follows:

$$\pi_t(\langle L,\, \rho,\, s\rangle) \stackrel{\text{def}}{=} \langle L(t),\, \rho\,[\forall t' \neq t : pc_{t'} \mapsto L(t')]\,,\, s^{-1}(t)\rangle \ . \tag{13}$$

Moreover, we distinguish two kinds of interferences in $(\mathcal{C} \times \mathcal{M}) \times (\mathcal{C} \times \mathcal{M})$:
– interferences from t that do not change the set M of mutexes held by t:

$$\begin{aligned}&\mathcal{I}^u(t)(M) \stackrel{\text{def}}{=} \\ &\quad \{\, \langle \langle L_i,\, \rho_i\rangle,\, \langle L_{i+1},\, \rho_{i+1}\rangle\rangle \mid \\ &\quad\quad \exists \langle L_0,\, \rho_0,\, s_0\rangle \xrightarrow{a_1} \cdots \langle L_n,\, \rho_n,\, s_n\rangle \in \mathcal{F} : a_i = t,\, s^{-1}_{i-1}(t) = s^{-1}_i(t) = M \,\}\end{aligned}$$

(a)

```
t1                          | t2
while random do             | while random do
  lock(m)                   |   lock(m)
  if $X < 100$ then         |   if $X < 100$ then
    $X \leftarrow X + 1$    |     $X \leftarrow X + 1$
  endif                     |   endif
  unlock(m)                 |   unlock(m)
done                        | done
```

(b)

```
t1                          | t2
while random do             | while random do
  lock(m)                   |   lock(m)
  if $X > 0$ then           |   if $X < 10$ then
    $X \leftarrow X - 1$    |     $X \leftarrow X + 1$
    $Y \leftarrow Y - 1$    |     $Y \leftarrow Y + 1$
  endif                     |   endif
  unlock(m)                 |   unlock(m)
done                        | done
```

Fig. 5. (a) Two identical threads concurrently incrementing a counter X protected by a lock m; and (b) an abstract producer/consumer with resources X and Y

– critical sections that summarize a sequence of transitions beginning with **lock**(m) by t, ending with **unlock**(m) by t, and containing transitions from any threads in-between:

$$\begin{aligned}&\mathcal{I}^s(t)(m) \stackrel{\text{def}}{=}\\ &\quad \{\, \langle\langle L_i,\, \rho_i\rangle,\, \langle L_j,\, \rho_j\rangle\rangle \mid \exists \langle L_0,\, \rho_0,\, s_0\rangle \xrightarrow{a_1} \cdots \langle L_n,\, \rho_n,\, s_n\rangle \in \mathcal{F} :\\ &\quad\; i < j,\, s_i(m) = s_j(m) = \bot,\, \forall k : i < k < j \implies s_k(m) = t \,\} \ .\end{aligned}$$

As in (6), the semantics of a thread t is computed by interleaving execution steps from the thread and from interferences. However, due to mutual exclusion, interferences in $\mathcal{I}^u(t')(M')$ cannot fire from a local state $\langle \ell,\, \rho,\, M\rangle$ of a thread $t \neq t'$ when $M \cap M' \neq \emptyset$. For instance, in Fig. 5.(a), no interference generated by $X \leftarrow X + 1$ in t_2 can run in t_1 between **lock**(m) and **unlock**(m). Moreover, mutual exclusion ensures that, if some interference in $\mathcal{I}^u(t')(M')$ such that $m \in M'$ fires from a local state $\langle \ell,\, \rho,\, M\rangle$ before t locks m, then t' must finish its critical section protected by m before t can lock m; hence, the interference is subsumed by a transition from $\mathcal{I}^s(t')(m)$. If the program is well-synchronized, i.e., every access to a variable V is protected by a mutex associated to the variable, then all the interferences are included in $\mathcal{I}^s(t')(m)$. Hence, it makes sense to abstract $\mathcal{I}^u(t')(M')$ in a coarse way, and use these interferences only in case (hopefully rare) of an unsynchronized access (i.e., a data race), while a more precise abstraction of $\mathcal{I}^s(t')(m)$ is used for well-synchronized accesses.

Following Sec. 3.2, we use a flow-insensitive and non-relational abstraction of $\mathcal{I}^u$, i.e.: $\mathcal{I}^{nr,u}(t')(M') \stackrel{\text{def}}{=} \alpha^{nr}_{\mathcal{I}}(\alpha^{nf}_{\mathcal{I}}(\mathcal{I}^u(t')(M')))$. By partitioning local thread states with respect to the program location $\ell \in \mathcal{L}$ and the mutexes held $M \in \mathcal{P}(\mathbb{M})$, and applying the optimization technique that an equation variable $\mathcal{X}_{\ell,M}$ represents a set of local states up to the pending interferences in $\mathcal{I}^{nr,u}(t')(M')$, assignments $V \leftarrow e$ give rise to equations of the form $\mathcal{X}_{\ell,M} = [\![\, V \leftarrow e \,]\!]\mathcal{X}_{\ell',M}$, where e is modified to incorporate all the interferences in $\mathcal{I}^{nr,u}(t')(M')$ such that $t' \neq t$ and $M \cap M' = \emptyset$ on variables appearing in e, and similarly for tests. An abstraction of the interferences in $\mathcal{I}^s(t')(m)$ is incorporated when t locks m. When $\langle \ell,\, \textbf{lock}(m),\, \ell'\rangle \in \mathit{inst}_t$, we generate, for each $M \in \mathcal{P}(\mathbb{M})$, an equation:

$$\mathcal{X}_{\ell',M\cup\{m\}} = \mathcal{X}_{\ell,M\setminus\{m\}} \cup \bigcup \{\, \mathit{apply}(\mathcal{I}^s(t')(m))(\mathcal{X}_{\ell,M\setminus\{m\}}) \mid t' \neq t \,\}$$

where the exact definition of *apply* depends on the abstraction chosen to approximate $\mathcal{I}^s$ and is discussed below. An **unlock**(m) instruction generates the simple equation: $\mathcal{X}_{\ell',M\setminus\{m\}} = \mathcal{X}_{\ell,M\cup\{m\}}$.

Example 5. As Fig. 5.(a) is well synchronized, the interference from $\mathcal{I}^{nr,u}$ on the assignment $X < 100$ and the test $X \leftarrow X + 1$ are empty. When choosing the flow-insensitive non-relational abstraction from Sec. 3.2 for $\mathcal{I}^s$ as well as $\mathcal{I}^u$, the interference caused by the critical section on both threads is $[X \mapsto [1, 100]]$, i.e., any value in $[1, 100]$ can be stored into X. The *apply* function is given by A^{nr}_t from (10). The resulting equation for **lock**(m) is thus: $\mathcal{X}_{\ell',\{m\}} = [\![\, X \leftarrow [1, 100] \,]\!] \mathcal{X}_{\ell,\emptyset} \cup \mathcal{X}_{\ell,\emptyset}$. This is sufficient to infer that X is always in $[0, 100]$. Recall that an analysis with the same non-relational abstraction but without interference partitioning would not find any bound on X.

To gain more precision, the invariance abstraction from Sec. 4.1 can be used for $\mathcal{I}^s$. The resulting analysis will infer relational properties of variables that are guaranteed to hold outside critical sections, but are possibly broken inside. We call them *lock invariants* by analogy with class invariants: in a well synchronized program, threads cannot observe program states where the invariant is broken by the action of another thread. Unlike the method of Sec. 4.1, we do not need to apply complex relational operations at all program points: the interferences are inferred by joining the environments only at **lock** and **unlock** instructions, while the *apply* function that incorporates interferences, given by R^{rel}_t (11), is only applied at **lock** instructions, which ensures an efficient analysis.

Example 6. Consider the program in Fig. 5.(b) that models an abstract producer/consumer, where X and Y denote the amount of resources. The non-relational interference analysis is sufficient to prove that X is bounded thanks to the explicit tests on X, but it cannot find any bound on Y. Using the invariant interference abstraction parameterized with the octagon domain [17], it is possible to infer that $X = Y$ is a lock invariant. This information automatically infers a bound on Y from the bound on X.

4.5 Weakly Consistent Memories

In the previous sections, we have assumed a sequentially consist model of execution [14]. Actually, computers may execute programs under more relaxed models, where different threads may hold inconsistent views of the memory [2], hence creating behaviors outside the sequentially consistent ones.

We justify informally the soundness of our interference analysis with respect to weakly consistent memories as follows: firstly, as proved in [19], flow-insensitive non-relational interference abstractions are naturally sound in a wide variety of memory models, hence our abstraction $\mathcal{I}^{nr,u}$ is sound; secondly, **lock** and **unlock** instructions provide memory synchronization points, so that any sound

monotonicity domain	relational lock invariants	analysis time	memory	iterations	alarms
×	×	25h 26mn	22 GB	6	4616
✓	×	30h 30mn	24 GB	7	1100
✓	✓	110h 38mn	90 GB	7	1009

Fig. 6. Experimental results for AstréeA on our 1.7 Mlines 15 threads code target

abstraction of $\mathcal{I}^s$ is also sound in relaxed models. Finally, the monotonicity abstractions proposed in Sec. 4.2 and Sec. 4.3 only rely on the ordering of sequences of assignments to each variable independently, and so, are sound in any memory model that guarantees it (such as the widespread Total Store Ordering).

5 Experimental Results

We have implemented our method in AstréeA, a static analyzer prototype [19] that extends Astrée. The Astrée analyzer [3] checks for run-time errors in embedded synchronous C programs. A specificity of Astrée is its specialization and design by refinement: starting from an efficient and coarse interval analyzer, we added new abstract domains until we reached the zero false alarm goal (i.e., a proof of absence of run-time error) on a pre-defined selection of target industrial codes, namely avionic control-command fly-by-wire software. The new abstractions are made tunable by end-users, to adapt the analysis to different codes in the same family. The result is an efficient and precise (few alarms) analyzer on general embedded C code, which is extremely precise (no alarm) on a restricted family of avionic embedded codes, and which is usable in industrial context [9].

The AstréeA prototype extends Astrée to analyze concurrent C programs. As Astrée, it does not support dynamic memory allocation nor recursivity (function calls are inlined for maximum precision) by design, as these are forbidden in most embedded platforms. It is also a specialized analyzer. Our main target is a large avionic code composed of 15 threads (without dynamic thread creation) totaling 1.7 Mlines of C and running under a real-time operating system based on the ARINC 653 specification; it performs a mix of numeric computations, reactive computations, network communications, and string formatting. More information on Astrée, AstréeA, ARINC 653, and our target application can be found in [3,20,19].

Using the design by refinement that made the success of Astrée, we started [19] with a simple analysis that reuses Astrée's sequential analysis and domains (including domains for machine integers, floats, pointers, relational domains, etc.), on top of which we added the simple non-relational abstraction of thread interferences from Sec. 3.2. The analysis time, peak memory consumption, as well as the number of iterations to stabilize interferences and the number of alarms are reported in the first line of Fig. 6. The framework presented in this article was developed when it became clear that non-relational interferences were too coarse to infer the properties needed to remove some false alarms (an example of which was given in Fig. 4). The second line of Fig. 6 presents experimental results after adding the monotonicity and subsequence domains of Sec. 4.2 and Sec. 4.3, while the last line also includes the relational lock invariants from Sec. 4.4. The monotonicity domain provides a huge improvement in precision for a reasonable cost; this is natural as it is a specialized domain designed to handle very specific uses of clocks and counters in our target application. The relational lock invariant domain can remove a few extra alarms, but it is not as well tuned and efficient yet: for now, it inherits without modification Astrée's relational domains and packing strategies (i.e., choosing *a priori* which variables to track in a relational way). Nonetheless, relational lock invariants are versatile and general purpose by nature; we believe that, by parameterizing them in future work with adequate relational domains and packing strategies, they have the potential to further improve the precision at a more reasonable cost.

Implementation-wise, adding these new domains did not require a large effort; in particular, the overall architecture of AstréeA and existing domains required only marginal changes. We benefited from casting the former analysis as an abstraction of a more concrete semantics, from which alternate abstractions could be derived and combined under a unified thread-modular analysis framework.

6 Conclusion

We have proposed a framework to design thread-modular static analyses that are able to infer and use relational and history-sensitive properties of thread interferences. This was achieved by a reinterpretation of Jones' rely-guarantee proof method as a constructive fixpoint semantics, which is complete for safety properties and can be abstracted into static analyses in a systematic way, thus following the abstract interpretation methodology. We then proposed several example abstractions tailored to solve specific problems out of the reach of previous, non-relational interference abstractions, and motivated by actual analysis problems. We presented encouraging results on the analysis of an embedded industrial C code using the AstréeA prototype analyzer.

AstréeA is very much a work in progress, and further work is required in order to improve its precision (towards the zero false alarm goal) and widen its application scope (to analyze more classes of embedded concurrent C software and hopefully enable a deployment in industry). This will require the design of new abstractions, in particular to improve our relational lock invariant inference. Another interesting promising area is the development of trace-related abstractions, with potential generalization to inferring maximal trace properties, which includes liveness properties, in a thread-modular way.

References

1. Amjad, H., Bornat, R.: Towards automatic stability analysis for rely-guarantee proofs. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 14–28. Springer, Heidelberg (2009)
2. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL 2010, pp. 7–18. ACM (January 2010)

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 2003, pp. 196–207. ACM (June 2003)
4. Carré, J.-L., Hymans, C.: From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS (October 2009)
5. 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)
6. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: ISP 1976, pp. 106–130, Dunod, Paris (1976)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM (January 1977)
8. Cousot, P., Cousot, R.: Invariance proof methods and analysis techniques for parallel programs. In: Automatic Program Construction Techniques, ch. 2, pp. 243–271. Macmillan, New York (1984)
9. Delmas, D., Souyris, J.: Astrée: From research to industry. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 437–451. Springer, Heidelberg (2007)
10. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theoretical Computer Science 338(1-3), 153–183 (2005)
11. 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)
12. Jeannet, B.: Relational interprocedural verification of concurrent programs. Software & Systems Modeling 12(2), 285–306 (2013)
13. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University (June 1981)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. on Computers 28, 690–691 (1979)
15. 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)
16. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
17. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
18. Miné, A.: Static analysis by abstract interpretation of sequential and multi-thread programs. In: MOVEP 2012, pp. 35–48 (December 2012)
19. Miné, A.: Static analysis of run-time errors in embedded real-time parallel C programs. Logical Methods in Computer Science 8(26), 63 (2012)
20. Miné, A.: Static analysis by abstract interpretation of concurrent programs. Habilitation report, École normale supérieure (May 2013)
21. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6(4), 319–340 (1976)
22. Rinard, M.: Analysis of multithreaded programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 1–19. Springer, Heidelberg (2001)
23. Watkins, C.B., Walter, R.: Transitioning from federated avionics architectures to integrated modular avionics. In: DASC 2007, vol. 2.A.1, pp. 1–10. IEEE (October 2007)

Timing Analysis of Parallel Software Using Abstract Execution

Andreas Gustavsson, Jan Gustafsson, and Björn Lisper

Mälardalen University, Västerås, Sweden
{andreas.sg.gustavsson,jan.gustafsson,bjorn.lisper}@mdh.se

Abstract. A major trend in computer architecture is multi-core processors. To fully exploit this type of parallel processor chip, programs running on it will have to be parallel as well. This means that even hard real-time embedded systems will be parallel. Therefore, it is of utmost importance that methods to analyze the timing properties of parallel real-time systems are developed.

This paper presents an algorithm that is founded on abstract interpretation and derives safe approximations of the execution times of parallel programs. The algorithm is formulated and proven correct for a simple parallel language with parallel threads, shared memory and synchronization via locks.

Keywords: WCET, Parallelism, Multi-core, Abstract interpretation, Abstract execution.

1 Introduction

A *real-time* system is a system for which the timing behavior is of great importance. *Hard* real-time systems are such that failure to produce the computational result within certain timing bounds could have catastrophic consequences. One example of a hard real-time system is the airbag system in automotive vehicles, another is the control system in airplanes.

A major trend in computer hardware design is *multi-core* processors. The processor cores on such a chip typically share some resources, such as some level of on-chip cache memory, which introduces dependencies and conflicts between the cores. Processor chips of this kind are already (and will, in the future, be even more extensively) incorporated in real-time systems.

To fully utilize the multi-core architecture, algorithms will have to be parallelized over multiple tasks (e.g. threads). This means that the tasks will have to share resources and communicate and synchronize with each other. There already exist software libraries for explicitly parallelizing sequential code automatically. One example of such a library available for C/C++ and Fortran code running on shared-memory machines is OpenMP [1]. The conclusion is that parallel software running on parallel hardware is already available today and will probably be the standard way of computing in the future, also for real-time systems. Thus, it is of crucial importance that methods to derive safe estimations on the lower and upper bounds of the execution times (also referred to as the Best-Case and Worst-Case Execution Times – BCET and WCET– respectively: see [2]) of parallel systems are derived.

This paper presents a novel method that derives safe estimations on the timing bounds for parallel software. The method mainly targets hard real-time systems but can be applied to any computer system that can be modeled using the presented method. More specifically, the main contributions of this paper are the following.

1. A formally defined parallel programming language (PPL) with shared memory, locks, and a timing model.
2. An algorithm that derives safe approximations of the BCET and WCET of PPL programs.

The rest of the paper is organized as follows. Section 2 describes the ideas behind abstract execution for sequential programs. Section 3 presents some research related to the method presented in this paper. Section 4 presents PPL, a parallel programming language. Section 5 abstractly interprets the semantics of PPL. Section 6 presents an algorithm that abstractly executes PPL programs to find safe approximations of their timing behaviors. Section 7 uses the presented algorithm to derive safe bounds on the BCET and WCET for an example PPL program given a simple timing model. Section 8 concludes the paper and presents directions for future research.

2 Abstract Execution for Sequential Programs

Abstract execution (AE) [3,4] was originally designed as a method to derive program flow constraints on imperative sequential programs, like bounds on the number of iterations in loops and infeasible program path constraints. This information can be used by a subsequent *WCET analysis* [2] to compute a safe WCET bound. AE is based on abstract interpretation, and is basically a very context sensitive value analysis which can be seen as a form of symbolic execution [3]. The program is hence executed in the abstract domain; i.e. abstract versions of the program operators are executed and the program variables have abstract values (which thus correspond to sets of concrete values).

The main difference between AE and a traditional value analysis is that in the former, an abstract state is not calculated for each program point. Instead, the abstract state is propagated on transitions in a way similar to the concrete state for concrete executions of the program. Note that since values are abstracted, a state can propagate to several new states on a single transition (e.g. when both branches of a conditional statement could be taken given the abstract values of the program variables in the current abstract state). Therefore, a worklist algorithm that collects all possible transitions is needed to safely approximate all concrete executions. There is a risk that AE does not terminate (e.g. due to an infinite loop in the analyzed program): however, if it terminates then all final states of the concrete executions have been safely approximated [3]. Furthermore, nontermination can be completely dealt with by setting a timeout, e.g. as an upper limit on the number of abstract transitions.

If timing bounds on the statements of the program are known, then AE is easily extended to calculate BCET and WCET bounds by treating time as a regular program variable that is updated on each state transition – as with all other variables, its set of possible final values is then safely approximated when the algorithm terminates [5].

The approach used in this paper is to calculate safe BCET and WCET estimations by abstract execution of the analyzed program. The timing bounds are derived based on a safe timing model of the underlying architecture.

3 Related Work

WCET-related research started with the introduction of timing-schemas by Shaw in 1989 [6]. Shaw presents rules to collapse the CFG (Control Flow Graph) of a program until a final single value represents the WCET. An excellent overview of the field of WCET research was presented by Wilhelm et al. in 2008 [2]. The field of WCET analysis for parallel software is quite new, so there is no solid foundation of previous research.

Model-checking has been shown adequate for timing analysis of small parts of single-core systems [7,8]. There are also attempts to analyze parallel systems using model-checking [9,10,11]. However, complexity matters is a common big issue for these attempts.

This paper uses a more approximate approach (abstract execution). If analyzing a program consisting of only one thread, the method presented in this paper becomes comparable to the methods presented by Gustafsson et al. [4] and Ermedahl et al. [5]. An early version of the analysis presented here [12] could analyze a subset of PPL (without locks); the version here can analyze any PPL program.

There are several other approaches toward WCET analysis of parallel and concurrent programs that are not defined based on abstract execution. Mittermayr and Blieberger [13] use a graph based approach and Kronecker algebra to calculate an estimation on the WCET of a concurrent program. Potop-Butucaru and Puaut [14] target static timing analysis of parallel processors where “channels” are used to communicate between, and synchronize, the parallel tasks. The goal of this approach is to enable the use of the traditional abstract interpretation techniques for sequential software when analyzing parallel systems. Ozaktas et al. [15] focus on analyzing synchronization delays experienced by tasks executing on time-predictable shared-memory multi-core architectures.

The work presented in these publications targets parallel systems with quite specific restrictions, whereas our analysis targets general parallel systems. We focus on analyzing parallel systems on code level, where the underlying architecture could be sequential or parallel, bare metal or an operating system. The only assumption is that the temporal behavior of the underlying architecture, and thus of the threads in the analyzed program, can be safely approximated.

4 PPL: A Parallel Programming Language

In this section, a rudimentary, parallel programming language, PPL, whose semantics includes timing behavior, will be presented. The purpose of PPL is to put focus on communication through shared memory and synchronization on shared resources.

The parallel entities of execution will be referred to as threads. PPL provides both thread-private and globally shared memory, referred to as registers, $r \in$ Reg, and variables, $x \in$ Var, respectively. Arithmetical operations etc. within a thread can be performed using the values of the thread's registers. PPL also provides shared resources, referred to as locks, $lck \in$ Lck, that can be acquired in a mutually exclusive manner by the threads. The operations (statements) provided by the instruction set may have variable execution times. (C.f. multi-core CPUs, which have both local and global memory, a shared memory bus and atomic, i.e. mutually exclusive, operations.) Note that Reg, Var and Lck are finite sets of identifiers that are specific to each defined PPL program.

The syntax of PPL, which is a set of operations using the discussed architectural features, is defined in Fig. 1. Π denotes a program, which simply is a (constant and finite) set of threads, i.e. $\Pi =$ Thrd, where each thread, $\mathrm{T} \in$ Thrd, is a pair of a unique identifier, $d \in \mathbb{Z}$, and a statement, $s \in$ Stm. This makes every thread unique and distinguishable from other threads, even if several threads consist of the same statement. The axiom-statements (all statements except the sequentially composed statement, $s_1\,;s_2$) of each thread are assumed to be uniquely labeled with consecutive labels, $l \in \mathbb{Z}$. $a \in$ Aexp and $b \in$ Bexp denote an arithmetic and a boolean expression, respectively, and $n \in \mathbb{Z}$ is an integer value.

$$
\begin{aligned}
\Pi &::= \{\mathrm{T}_1, \dots, \mathrm{T}_m\} \\
\mathrm{T} &::= (d, s) \\
s &::= [\texttt{halt}]^l \;\big|\; [\texttt{skip}]^l \;\big|\; [r := a]^l \;\big|\; [\texttt{if}\ b\ \texttt{goto}\ l']^l \;\big|\; [\texttt{store}\ r\ \texttt{to}\ x]^l \;\big| \\
&\qquad [\texttt{load}\ r\ \texttt{from}\ x]^l \;\big|\; [\texttt{unlock}\ lck]^l \;\big|\; [\texttt{lock}\ lck]^l \;\big|\; s_1\,;s_2 \\
a &::= n \;\big|\; r \;\big|\; a_1 \texttt{+} a_2 \;\big|\; a_1 \texttt{-} a_2 \;\big|\; a_1 \texttt{*} a_2 \;\big|\; a_1 \texttt{/} a_2 \\
b &::= \texttt{true} \;\big|\; \texttt{false} \;\big|\; \texttt{!}b \;\big|\; b_1 \,\texttt{\&\&}\, b_2 \;\big|\; a_1 \texttt{==} a_2 \;\big|\; a_1 \texttt{<=} a_2
\end{aligned}
$$

Fig. 1. Syntax of the parallel programming language, PPL

The semantic state of a program is described by a configuration, c, defined as $\langle [\mathrm{T}, pc_\mathrm{T}, \mathbb{r}_\mathrm{T}, t^a_\mathrm{T}]_{\mathrm{T}\in\mathrm{Thrd}}, \mathbb{x}, \mathbb{l}\rangle$. The notation $[\mathrm{T}, pc_\mathrm{T}, \mathbb{r}_\mathrm{T}, t^a_\mathrm{T}]_{\mathrm{T}\in\{\mathrm{T}_1,\dots,\mathrm{T}_m\}}$ expands to $\langle \mathrm{T}_1, pc_{\mathrm{T}_1}, \mathbb{r}_{\mathrm{T}_1}, t^a_{\mathrm{T}_1}, \dots, \mathrm{T}_m, pc_{\mathrm{T}_m}, \mathbb{r}_{\mathrm{T}_m}, t^a_{\mathrm{T}_m}\rangle$. This notation is needed since the number of threads in a program is not known before the program is defined.

pc_T is a program counter, pointing to the current statement in T. Note that the tuple $\langle pc_{\mathrm{T}_1}, \dots, pc_{\mathrm{T}_m}\rangle$, assuming that Thrd $= \{\mathrm{T}_1, \dots, \mathrm{T}_m\}$, defines a unique program point. $\mathbb{r}_\mathrm{T}$ is a mapping from T's registers to their values. t^a_T is the accumulated execution time of T. $\mathbb{x}$ is a mapping from variables and threads to a set of timestamped values. $\mathbb{l}$ is a mapping from locks to their states. The state for a lock is a tuple containing information on (in the following order) whether the lock is acquired or not, which thread owns it, a deadline for when the lock must be acquired by the owning thread, the previous owner, and when it was last released. (If the reader finds the variable and lock domains peculiar, the need for their definitions will become clear in Sects. 5 and 6.) The BCET and WCET for a set of configurations are given in Definition 1.

Definition 1. *Given a set of configurations, C, the* BCET *and* WCET *for that set are defined as:*

$$\begin{cases} \text{BCET} ::= \min(\{\max(\{t^a_\text{T} \mid \text{T} \in \text{Thrd}\}) \mid \langle [\text{T}, pc_\text{T}, \mathbb{r}_\text{T}, t^a_\text{T}]_{\text{T}\in\text{Thrd}}, \mathbb{x}, \mathbb{l}\rangle \in C\}) \\ \text{WCET} ::= \max(\{\max(\{t^a_\text{T} \mid \text{T} \in \text{Thrd}\}) \mid \langle [\text{T}, pc_\text{T}, \mathbb{r}_\text{T}, t^a_\text{T}]_{\text{T}\in\text{Thrd}}, \mathbb{x}, \mathbb{l}\rangle \in C\}) \end{cases}$$

The semantics of transitions between configurations is described by $\xrightarrow[prg]{}$ as defined in Fig. 2. *exp1 ? exp2 : exp3* is *exp2* if *exp1*, and *exp3* otherwise. $\lambda p \in P.exp(p)$ is a function from p (an element of P) to $exp(p)$. STM gives the current statement of the issuing thread. TIME gives a relative execution time for the current statement of the calling thread (the definition of this function is out of this paper's scope, but it is assumed to be non-negative). Given some lock state mapping and some lock, OWN gives the owner of the lock (which is $\bot_{thrd}$ iff the lock is free; note however that Thrd is not a complete lattice), POWN gives the previous owner of the lock, REL gives the time at which the lock was last released. Note that similar functions can be defined to mask out the current state – *taken* or *free* – and the lock owner assignment deadline [16].

$\xrightarrow[ax]{}$ (whose formal definition is omitted due to space limitations; see [16]) describes the semantics of a single statement within a thread when considered in isolation from other threads: `halt` stops the execution of the issuing thread (i.e. none of the input states are changed), `halt` must be the last statement of each thread in the program, but could also occur anywhere "within" a thread; `skip` performs a no-operation (i.e. it only increments the thread's program counter); a register is assigned a value using := (the semantics of evaluating arithmetic and boolean expressions are defined in the standard fashion [16,17] and will not be further discussed); conditional branching to an arbitrary axiom-statement is performed using `if` (thus, `if` is used when e.g. implementing loops); `store` makes the thread's set of timestamped values for the given variable consist only of a tuple consisting of the value of the given register and the value of $t^{a\prime}_\text{T}$ (i.e. t); `load` takes one of the stored timestamped values for the given variable (after any `store`, there is only one such value for the given variable since only one of the values stored to a variable by a set of threads is saved; c.f. Fig. 2) and puts the value into the given register; `unlock` releases the given lock (i.e. sets the lock's state to *free*, its owner to $\bot_{thrd}$, its previous owner to the issuing thread, and its release time to t) if the issuing thread is the owner of the lock, otherwise `unlock` is a no-operation; `lock` is used to acquire the given lock in a mutually exclusive manner. Note that $\mathbb{l}''$ is used to choose which thread in a set

$$\frac{\mathrm{Thrd}_{exe} \neq \emptyset \wedge \forall \mathrm{T} \in \mathrm{Thrd}_{exe} : \langle \mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, \mathbb{x}, \mathbb{l}'', t^{a\prime}_{\mathrm{T}} \rangle \xrightarrow[ax]{} \langle pc'_{\mathrm{T}}, \mathbb{r}'_{\mathrm{T}}, \mathbb{x}'_{\mathrm{T}}, \mathbb{l}'_{\mathrm{T}} \rangle}{\begin{array}{l} c = \langle [\mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, t^a_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \mathbb{x}, \mathbb{l} \rangle \xrightarrow[prg]{} \\ \quad c' = \langle [\mathrm{T}, (\mathrm{T} \in \mathrm{Thrd}_{exe} \ ?\ pc'_{\mathrm{T}} : pc_{\mathrm{T}}), (\mathrm{T} \in \mathrm{Thrd}_{exe} \ ?\ \mathbb{r}'_{\mathrm{T}} : \mathbb{r}_{\mathrm{T}}), t^{a\prime}_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \mathbb{x}', \mathbb{l}' \rangle \end{array}}$$

where

$$t = \min(\{ t^a_{\mathrm{T}} + \textsc{time}(c, \mathrm{T}) \mid \mathrm{T} \in \mathrm{Thrd} \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\mathtt{halt}]^{pc_{\mathrm{T}}} \})$$

$$\mathrm{Thrd}_{exe} = \{ \mathrm{T} \in \mathrm{Thrd} \mid t = t^a_{\mathrm{T}} + \textsc{time}(c, \mathrm{T}) \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\mathtt{halt}]^{pc_{\mathrm{T}}} \}$$

$$t^{a\prime}_{\mathrm{T}} = \begin{cases} t^a_{\mathrm{T}} + \textsc{time}(c, \mathrm{T}) & \textbf{if } \mathrm{T} \in \mathrm{Thrd}_{exe} \\ t^a_{\mathrm{T}} & \textbf{otherwise} \end{cases}$$

$$\mathbb{x}'\ x = \begin{cases} \mathbb{x}\ x & \textbf{if } \mathrm{Thrd}_x = \emptyset \\ \lambda \mathrm{T} \in \mathrm{Thrd}.(\mathrm{T} = \mathrm{T}' \ ?\ (\mathbb{x}'_{\mathrm{T}'}\ x)\ \mathrm{T}' : \emptyset) & \textbf{otherwise} \\ \qquad \textbf{where } \mathrm{T}' \text{ is one of the threads in } \mathrm{Thrd}_x = & \\ \quad \{ \mathrm{T} \in \mathrm{Thrd}_{exe} \mid \exists r \in \mathrm{Reg}_{\mathrm{T}} : \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\mathtt{store}\ r\ \mathtt{to}\ x]^{pc_{\mathrm{T}}} \} & \end{cases}$$

$$\mathbb{l}''\ lck = \begin{cases} (\mathit{free}, \mathrm{T}', t, & \textbf{for some } \mathrm{T}' \in \{ \mathrm{T} \in \mathrm{Thrd}_{exe} \mid \\ \quad \textsc{pown}(\mathbb{l}\ lck), & \qquad \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\mathtt{lock}\ lck]^{pc_{\mathrm{T}}} \}, \\ \qquad \textsc{rel}(\mathbb{l}\ lck)) & \textbf{if } \{ \mathrm{T} \in \mathrm{Thrd}_{exe} \mid \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = \\ & \qquad [\mathtt{lock}\ lck]^{pc_{\mathrm{T}}} \} \neq \emptyset \wedge \textsc{own}(\mathbb{l}\ lck) = \bot_{thrd} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}'\ lck = \begin{cases} \mathbb{l}'_{\textsc{own}(\mathbb{l}''\ lck)}\ lck & \textbf{if } \textsc{own}(\mathbb{l}''\ lck) \in \mathrm{Thrd}_{exe} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

Fig. 2. $c \xrightarrow[prg]{} c'$, the semantics of concrete transitions

of competing threads is successful in acquiring the lock and that the unsuccessful threads wait in a spin-lock fashion until the lock is released – which means that configurations can deadlock.

It should be apparent that the threads included in a transition between two configurations (i.e. the threads included in Thrd_{exe}) are such that they execute their respective current statement at the earliest point in time at which any such event occurs (i.e. at t). When a thread issuing `lock` is assigned the given lock, it sets the lock's state to *taken*. As can be seen, the lock assignment deadline is always t; i.e. the time at which a `lock`-statement is issued on the free lock and a lock owner assignment occurs (which means that the deadline will always be met by the assigned lock owner since the owner is guaranteed to be one of the threads in Thrd_{exe} that issue a `lock`-statement on the given lock – which also means that the state of the lock is *taken* iff the owner of it is not $\bot_{thrd}$). The complete semantics of PPL is formally defined and more extensively discussed in [16].

5 Abstractly Interpreting PPL

In the following, time will be assumed to be abstractly interpreted as an interval (c.f. [17]). For simplicity, values are also abstractly interpreted using the interval domain. However, several other domains for values could be used instead.

The abstract semantic state of a program is described by an abstract configuration, $\tilde{c} = \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle$. Like for the concrete configuration, pc_{T} is a program counter, pointing to the current statement in T. $\tilde{\mathbb{r}}_{\mathrm{T}}$ is a mapping (i.e. a function) from T's registers to their abstract values (i.e. intervals). $\tilde{t}^a_{\mathrm{T}}$ is the accumulated execution time of T (i.e. an interval). $\tilde{\mathbb{x}}$ is a mapping from variables and threads to a set of timestamped values (i.e. pairs of intervals), where each such value might represent the actual value stored to the variable at the interval in time represented by $\tilde{c}$. $\tilde{\mathbb{l}}$ is a mapping from locks to tuples containing information on (in the following order) whether the lock is acquired or not, which thread owns it, a deadline for when the lock must be acquired by the owning thread, the previous owner, and when it was last released. It should thus be apparent that an abstract configuration corresponds to a set of concrete configurations. Thus, these domains safely over-approximate the corresponding concrete domains (Lemma 1).

Lemma 1. *An abstract configuration safely approximates a set of concrete configurations.*

Proof (sketch). This proof is conducted by first showing that there are Galois Connections [17] between the concrete and abstract domains for register, variable and lock mappings. Note that since the interval domain is used to approximate values and times, there exist Galois Connections between the concrete and abstract domains for values and time [17]. Finally, it is shown that there is a Galois Connection between the configuration domains [16]. □

From Definition 1, it is easy to see that for the interval domain, the BCET and WCET must be as given by Definition 2. α_t and γ_t are the abstraction and concretization functions between the concrete time and abstract time (i.e. interval) domains [17].

Definition 2. *Given a set of abstract configurations, $\tilde{C}$, the concrete* BCET *and* WCET *for that set are defined as:*

$$\begin{cases} \mathrm{BCET} ::= \min(\{\max(\{\min(\gamma_t(\tilde{t}^a_{\mathrm{T}})) \mid \mathrm{T}\in\mathrm{Thrd}\}) \mid \\ \qquad\qquad \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \in \tilde{C}\}) \\ \mathrm{WCET} ::= \max(\{\max(\{\max(\gamma_t(\tilde{t}^a_{\mathrm{T}})) \mid \mathrm{T}\in\mathrm{Thrd}\}) \mid \\ \qquad\qquad \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \in \tilde{C}\}) \end{cases}$$

The semantics of transitions between abstract configurations is described by $\xrightarrow[prg]{\sim}$ as defined in Fig. 3. Note that: ABSTIME, although its definition is out of scope for this paper, is assumed to be a safe approximation of TIME (Assumption 1); DLLOCK gives a safe approximation of the concrete point in time when the given lock must be acquired by some thread [16]; ACCTIME, considering some thread, T, gives a safe approximation of $t^{a\prime}_{\mathrm{T}}$ as defined in Fig. 2 [16]; $\widetilde{\mathrm{OWN}}$, $\widetilde{\mathrm{POWN}}$ and $\widetilde{\mathrm{REL}}$ are the abstract counterparts of the masking functions OWN, POWN and REL, respectively. (The definitions of the above functions are omitted due to space limitations; see [16].) $i_1 \tilde{+}_t i_2$ is the sum of the two intervals i_1 and i_2 [16]. $\xrightarrow[ax]{\sim}$ is further discussed below.

$$\frac{\mathrm{Thrd}_{exe} \neq \emptyset \wedge \forall \mathrm{T} \in \mathrm{Thrd}_{exe} : \langle \mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}'', \tilde{t}^{a\prime}_{\mathrm{T}}\rangle \xrightarrow[ax]{\sim} \langle pc'_{\mathrm{T}}, \tilde{\mathbb{r}}'_{\mathrm{T}}, \tilde{\mathbb{x}}'_{\mathrm{T}}, \tilde{\mathbb{l}}'_{\mathrm{T}}\rangle}{\begin{array}{l} \tilde{c} = \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^{a}_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \xrightarrow[prg]{\sim} \\ \quad \tilde{c}' = \langle [\mathrm{T}, (\mathrm{T} \in \mathrm{Thrd}_{exe} \; ? \; pc'_{\mathrm{T}} : pc_{\mathrm{T}}), (\mathrm{T} \in \mathrm{Thrd}_{exe} \; ? \; \tilde{\mathbb{r}}'_{\mathrm{T}} : \tilde{\mathbb{r}}_{\mathrm{T}}), \tilde{t}^{a\prime}_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}', \tilde{\mathbb{l}}'\rangle \end{array}}$$

where

$$\tilde{t}^{r}_{\mathrm{T}} = \textsc{absTime}(\tilde{c}, \mathrm{T})$$

$$\tilde{t}_{all} = \alpha_t(\{\min(\{\min(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid B\}), \min(\{\max(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid B\})\})$$

where $B \iff \mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\mathtt{halt}]^{pc_{\mathrm{T}}} \wedge \forall lck \in \mathrm{Lck} : (\textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\mathtt{lock}\ lck]^{pc_{\mathrm{T}}} \Rightarrow \tilde{\textsc{own}}(\tilde{\mathbb{l}}\ lck) \in \{\bot_{thrd}, \mathrm{T}\})$

$$\mathrm{Thrd}^{all}_{exe} = \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \mid \tilde{t}_{all} \tilde{\sqcap}_t (\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}}) \neq \tilde{\bot}_t \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\mathtt{halt}]^{pc_{\mathrm{T}}}\}$$

$$\tilde{\mathbb{l}}''\ lck = \begin{cases} (free, \mathrm{T}', & \textbf{for some } \mathrm{T}' \in \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \mid \\ \quad \textsc{dlLock}(\tilde{c}, lck), & \qquad \exists l \in \mathbb{Z} : \textsc{stm}(\mathrm{T}, l) = [\mathtt{lock}\ lck]^{l}\}, \\ \quad \tilde{\textsc{pown}}(\tilde{\mathbb{l}}\ lck), & \textbf{if } \exists \mathrm{T} \in \mathrm{Thrd}^{all}_{exe} : \tilde{\textsc{own}}(\tilde{\mathbb{l}}\ lck) = \bot_{thrd} \wedge \\ \quad \tilde{\textsc{rel}}(\tilde{\mathbb{l}}\ lck)) & \qquad \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\mathtt{lock}\ lck]^{pc_{\mathrm{T}}} \\ \tilde{\mathbb{l}}\ lck & \textbf{otherwise} \end{cases}$$

$$\mathrm{Thrd}_{\mathbf{hold}} = \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \mid \exists lck \in \mathrm{Lck} : (\textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\mathtt{lock}\ lck]^{pc_{\mathrm{T}}} \wedge \tilde{\textsc{own}}(\tilde{\mathbb{l}}''\ lck) \neq \mathrm{T}) \vee \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\mathtt{halt}]^{pc_{\mathrm{T}}}\}$$

$$\tilde{t} = \alpha_t(\{\min(\{\min(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid \mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \setminus \mathrm{Thrd}_{\mathbf{hold}}\}), \min(\{\max(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid \mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \setminus \mathrm{Thrd}_{\mathbf{hold}}\})\})$$

$$\mathrm{Thrd}_{exe} = \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \setminus \mathrm{Thrd}_{\mathbf{hold}} \mid \tilde{t} \tilde{\sqcap}_t (\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}}) \neq \tilde{\bot}_t\}$$

$$\tilde{\mathbb{l}}'\ lck = \begin{cases} \tilde{\mathbb{l}}'_{\tilde{\textsc{own}}(\tilde{\mathbb{l}}''\ lck)}\ lck & \textbf{if } \tilde{\textsc{own}}(\tilde{\mathbb{l}}''\ lck) \in \mathrm{Thrd}_{exe} \\ \tilde{\mathbb{l}}''\ lck & \textbf{otherwise} \end{cases}$$

$$\tilde{\mathbb{x}}' = \begin{cases} \textsc{trim}(\tilde{\mathbb{x}}'', \tilde{t}) & \textbf{if } \mathrm{Thrd}_{\tilde{c}} = \mathrm{Thrd} \\ \tilde{\mathbb{x}}'' & \textbf{otherwise} \end{cases}$$

where $(\tilde{\mathbb{x}}''\ x)\ \mathrm{T} = \begin{cases} (\tilde{\mathbb{x}}'_{\mathrm{T}}\ x)\ \mathrm{T} & \textbf{if } \mathrm{T} \in \mathrm{Thrd}_{exe} \\ (\tilde{\mathbb{x}}\ x)\ \mathrm{T} & \textbf{otherwise} \end{cases}$

$$\tilde{t}^{a\prime}_{\mathrm{T}} = \textsc{accTime}(\langle [\mathrm{T}', pc_{\mathrm{T}'}, \tilde{\mathbb{r}}_{\mathrm{T}'}, \tilde{t}^{a}_{\mathrm{T}'}]_{\mathrm{T}'\in\mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}''\rangle, \mathrm{Thrd}_{exe}, \mathrm{T})$$

Fig. 3. $\tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'$, semantics of abstract transitions

Assumption 1. *It is assumed that* ABSTIME *is a "non-negative" function in the interval domain that safely approximates* TIME *for any thread in any configuration, given a specific value of the thread's program counter, at a specific point (interval) in time.*

Like in the concrete semantics, which threads that execute their respective current statement on a given abstract transition is determined based on when in time this would happen. However, since time is approximated using intervals, it might not be possible to determine the exact order in which certain events occur in the abstract case:

1. The sets of threads that will execute their current statements on a transition (i.e. Thrd_{exe}) might differ between the concrete and abstract cases even if the given concrete configuration is safely approximated by the abstract one. Because of this, different program points might be “visited” in the concrete and abstract cases, and thus, the concrete collecting semantics (i.e. all configurations that are reachable from a set of initial configurations [18,3]) cannot be safely over-approximated using $\xrightarrow[prg]{\sim}$.
2. The execution of `load`-statements cannot be safely approximated using the semantic transition rules if the `load` is not the sole statement executed in the transition and the value of a global variable (i.e. a variable that might be read by at least one thread and that might be written by at least one other thread) is to be loaded. The reason for this is that other threads might execute `store`-statements, writing to the loaded variable, in succeeding transitions that could semantically occur before the `load`-statement in the concrete case.
3. A similar reasoning to that for `load`-statements holds for `lock`-statements; a non-acquired lock cannot simply be assigned to one of the threads in Thrd_{exe} that are trying to acquire it, because in the concrete case, some other thread might be the first to acquire the lock.
4. Since threads are spinning on locks that are owned by some other thread in the concrete case, but are frozen (see below) in the abstract case, the timing behavior of deadlocked transitions cannot be safely approximated.

$\xrightarrow[ax]{\sim}$ (whose formal definition is omitted due to space limitations; see [16]) describes the abstract semantics of a single statement within a thread when considered in isolation from other threads. There is no difference between the concrete and abstract behavior of the `halt`-, `skip`-, `:=`- and `unlock`-statements; however, the abstract semantics of evaluating arithmetic (and boolean) expressions is safely induced from the concrete semantics [16,17]. The abstract semantics of the `if`-statement is equivalent to the concrete semantics, with the exception that the register mapping for the issuing thread is restricted to exclude cases for which the given boolean expression cannot possibly hold [3,16]. `store` now adds a tuple consisting of the value of the given register (i.e. an interval) and the value of $\tilde{t}_{\mathrm{T}}^{a\prime}$ (i.e. an interval) to the issuing thread's (i.e. T's) set of timestamped values for the given variable. `load` now loads the given register of the issuing thread (i.e. T) with the least upper bound of all values that could be the actual value of the given variable at $\tilde{t}_{\mathrm{T}}^{a\prime}$. Note that TRIM removes timestamped values from the given variable mapping that will never affect a `load`-statement for the given variable in any thread at the given point (i.e. interval) in time or in the future [16]. `lock` still acquires the given lock only if the issuing thread is the owner of the lock (in $\tilde{\mathbb{I}}''$). A difference between the concrete and abstract semantics for `lock` is that whenever some thread issues a `lock`-statement on a free lock in the abstract case, any thread that might want to acquire the lock somewhere in the program could be assigned the ownership of the lock; note that this means that a lock can be owned by some thread without actually being acquired by that thread (i.e. the state of the lock is *free* even if the owner is not $\perp_{thrd}$). Another difference is that in the abstract case, the issuing thread will be frozen (not at all considered in transitions) if the given lock is owned by some other thread. The issuing thread remains frozen until the lock becomes free again.

If a `lock`-issuing thread has not already acquired the given lock, then it must be that $\tilde{t}_{\mathrm{T}}^{a\prime}$ has not passed the deadline for the lock owner assignment and that the release time of the lock is not in the future for `lock` to successfully acquire the lock. If $\tilde{t}_{\mathrm{T}}^{a\prime}$ has passed the deadline for the lock owner assignment, then the lock owner assignment, and thus the configuration, has no concrete counterpart since it must be that some other thread has already acquired the given lock [16]. If the lock's release time is in the future, then $\tilde{t}_{\mathrm{T}}^{a\prime}$ will be increased to safely approximate the concrete spin-waiting [16].

The abstract transitions described by $\xrightarrow[prg]{\sim}$ safely approximate the corresponding concrete transitions, for each thread individually, if they do not include the loading of a global variable in some thread and all threads wanting to acquire some lock are eventually able to do so (Lemma 2); i.e. if the hazards in the problems described by 2 and 4 above do not occur. The problems described by 1–4 above will be further discussed in the next section.

Lemma 2. *For each possible chain of transitions (as described by $\xrightarrow[prg]{}$) given some concrete configuration, there is an abstract chain of transitions (as described by $\xrightarrow[prg]{\sim}$) that safely approximates the concrete chain, for each thread considered individually, given that the initial abstract configuration safely approximates the initial concrete configuration, the thread is eventually able to acquire any lock it wants to, and either $|\mathrm{Thrd}_{exe}| \not> 1$ or there is no thread in Thrd_{exe} that loads the value of a global variable for each transition on the chain.*

Proof (sketch). First note that the timing behavior of each thread can be considered in isolation from any other thread (follows from Assumption 1) and that $\xrightarrow[ax]{\sim}$ safely approximates $\xrightarrow[ax]{}$ (which partly follows from Lemma 1) [16].

Since either $|\mathrm{Thrd}_{exe}| \not> 1$ or there is no thread in Thrd_{exe} that loads the value of a global variable for each transition on the chain, it must be that the loading of global variables' values are never under-approximated since there cannot be any `store`-statements in any thread that can be issued in future transitions and that could semantically affect the loaded value [16].

And, since any thread that might want to acquire a lock somewhere in the program can become the owner of that lock when some thread issues a `lock`-statement on the given lock, and since all threads that want to acquire a lock will eventually be able to do so, it must be that there exist safe approximations of all concrete scenarios including synchronization on locks [16].

Thus, since $\xrightarrow[ax]{\sim}$ safely approximates $\xrightarrow[ax]{}$ and the timing behavior of each thread can be considered in isolation from any other thread, it must be that the lemma holds. □

6 Analyzing PPL Programs Using Abstract Execution

The abstract execution function, absExe, defined in Algorithm 1, is a worklist algorithm that encapsulates $\xrightarrow[prg]{\sim}$ and explicitly handles the problems discussed in the previous section. A configuration is said to be in the final state if all threads are issuing the `halt`-statement. A configuration is said to be deadlocked if it cannot possibly reach the final state according to the semantic transition rules. A configuration is said to be timed-out if the final state cannot possibly be reached before a given point in time (i.e. $\tilde{t}_{to}$) according to the semantic transition rules. A configuration is said to have valid concrete counterparts if it represents at least one concrete configuration that can semantically occur. Two cases for which a configuration lacks concrete counterparts are when a deadlock involves a non-acquired lock and when the owner of a non-acquired lock misses to acquire it before the expiration of the owner assignment deadline. Such configurations are discontinued. Note that a configuration representing a lock owner assignment where the owner of some lock has not yet acquired the lock, and the owner's accumulated execution time has not passed the owner assignment deadline, reaches a configuration with valid concrete counterparts if the owner issues a `lock`-statement on (i.e. acquires) the lock before the expiration of the deadline. A formal definition of ABSEXE is found in [16].

The overall strategy of the algorithm is depicted in Fig. 4 (α_{conf} and γ_{conf} are the abstraction and concretization functions for configurations, respectively [16]); i.e. given some safely approximated (by $\tilde{c}_0$) concrete configuration, c_0, there is an abstract transition sequence (which is safe for each thread individually) for each possible concrete transition sequence starting from c_0, and if the concrete sequence reaches a final state configuration (c_q), then so will the corresponding abstract sequence and the concrete final state configuration will be safely approximated (considering all threads) by the abstract final state configuration ($\tilde{c}_w$). Note that c_1, c_2, $\ldots$, c_{q-1} might not be safely approximated to their entirety by any of the abstract configurations $\tilde{c}_1$, $\tilde{c}_2$, $\ldots$, $\tilde{c}_{w-1}$ because of problem 1, defined on page 67. Although, it should be noted that for each thread individually, there are abstract configurations among these that safely approximate all the concrete states of that thread on the given concrete transition sequence.

For each thread that issues a `load`-statement on some global variable while not being the sole thread in Thrd_{exe}, ABSEXE removes that thread from the configuration and calls itself recursively (with an adapted timeout value) to derive all the possible values that could be loaded by the thread. Note that this is possible since the state for variables is a mapping from variables and threads to a set of timestamped values. This strategy addresses problem 2. Problem 3 is partly addressed in the definition of $\xrightarrow[prg]{\sim}$ (c.f. Fig. 3), as discussed in the previous section. ABSEXE fully addresses the problem by collecting all the possible transitions (i.e. resulting configurations) and adding them to the worklist. Problem 4 is addressed by identifying deadlocked configurations and aborting their transitions.

ABSEXE($\tilde{C}$, $\tilde{t}_{to}$) hence safely approximates the timing behavior of all threads in any configuration in the input set, $\tilde{C}$, up until $\tilde{t}_{to}$ (Theorem 1). It should be noted that if a transition sequence is aborted before a final state configuration is reached (e.g. because a deadlocked or timed-out configuration is identified), then an infinite WCET must be assumed for that transition sequence.

Algorithm 1. Abstract Execution

```
1: function absExe($\tilde{C}, \tilde{t}_{to}$)
2:     $\tilde{C}^w \leftarrow \tilde{C}$ ;   $\tilde{C}^f \leftarrow \emptyset$ ;   $\tilde{C}^d \leftarrow \emptyset$ ;   $\tilde{C}^t \leftarrow \emptyset$
3:     while $\tilde{C}^w \neq \emptyset$ do
4:         extract a configuration, $\tilde{c}$, from $\tilde{C}^w$
5:         $\tilde{C}^w \leftarrow \tilde{C}^w \setminus \{\tilde{c}\}$
6:         if $\tilde{c}$ is in the final state then
7:             $\tilde{C}^f \leftarrow \tilde{C}^f \cup \{\tilde{c}\}$
8:         else if $\tilde{c}$ is in a deadlocked state then
9:             $\tilde{C}^d \leftarrow \tilde{C}^d \cup \{\tilde{c}\}$
10:        else if $\tilde{c}$ is timed-out given $\tilde{t}_{to}$ then
11:            $\tilde{C}^t \leftarrow \tilde{C}^t \cup \{\tilde{c}\}$
12:        else if $\tilde{c}$ has, or could reach a $\tilde{c}'$ with, valid concrete counterparts then
13:            if a transition from $\tilde{c}$ includes more than one thread and some thread
                  would load the value of a global variable then
14:                for each thread, T, that loads global data in the transition from $\tilde{c}$
15:                    let $\tilde{c}^{\mathrm{T}}$ be like $\tilde{c}$, but with T and all its local states removed
16:                    let $\tilde{t}_{to}^{\mathrm{T}}$ be such that, after this time, the data can be safely loaded
17:                    let $(\tilde{C}_{\mathrm{T}}^f, \tilde{C}_{\mathrm{T}}^d, \tilde{C}_{\mathrm{T}}^t)$ be absExe($\{\tilde{c}^{\mathrm{T}}\}, \tilde{t}_{to}^{\mathrm{T}}$)
18:                    let T's loaded value be the least upper bound of the values that would
                       be loaded for all configurations in $\tilde{C}_{\mathrm{T}}^f \cup \tilde{C}_{\mathrm{T}}^d \cup \tilde{C}_{\mathrm{T}}^t \cup \{\tilde{c}\}$
19:                end for
20:                let $\tilde{c}'$ be like $\tilde{c}$, but with the loading of global data safely approximated
21:                $\tilde{C}^w \leftarrow \tilde{C}^w \cup \{\tilde{c}'\}$
22:            else
23:                $\tilde{C}^w \leftarrow \tilde{C}^w \cup \{\tilde{c}' \mid \tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'\}$
24:            end if
25:        end if
26:    end while
27:    return $(\tilde{C}^f, \tilde{C}^d, \tilde{C}^t)$
28: end function
```

Theorem 1. *For each final state configuration in the concrete collecting semantics, given some initial set of configurations, C,* absExe($\tilde{C}, \tilde{t}_{to}$) *derives either a safe approximation of that configuration or aborts the transition sequence at some point due to reaching a timed-out configuration with respect to $\tilde{t}_{to}$ whenever it terminates, given that $\forall c \in C : \exists \tilde{c} \in \tilde{C} : c \in \gamma_{conf}(\tilde{c})$. Likewise, for all configurations in the concrete collecting semantics that are deadlocked,* absExe *derives either a deadlocked or timed-out configuration, whenever it terminates.*

Proof (sketch). First note that

1. there are Galois Connections between all concrete and abstract domains, including the domains for configurations (Lemma 1),

2. all concrete transitions described by $\xrightarrow[prg]{}$ are safely approximated by $\xrightarrow[prg]{\sim}$, provided that whenever a thread issues a `load`-statement on a global variable, that thread is the sole thread in Thrd_{exe} (Lemma 2),
3. all possible transitions for a given configuration, as described by $\xrightarrow[prg]{\sim}$ are collected and added to the worklist (note that this safely approximates all concrete orders in which threads can be assigned the ownerships of locks – this solves the problem discussed in 3 in the previous section),
4. if a thread issues a `load`-statement on a global variable and that thread is not the sole thread in Thrd_{exe}, then it is easy to see that the **for each**-loop (i.e. the recursive use of ABSEXE) derives a safe approximation of the value as seen by that thread when issuing the `load`-statement (follows from Assumption 1 and Lemma 2) – this solves the problem discussed in 2 in the previous section,
5. the recursive calling of ABSEXE eventually stops since the set of threads in any PPL program is finite,
6. if any of the added configurations lacks (and cannot reach a configuration that has) valid concrete counterparts, it is trivially safe to discontinue it,
7. deadlocked transition sequences are aborted, but remembered – this solves the problem discussed in 4 in the previous section, and
8. timed-out transition sequences are aborted, but remembered.

It is thus easy to see that the combined use of $\xrightarrow[prg]{\sim}$ and the explicit handling of each thread loading the value of a global variable when that thread is not alone in Thrd_{exe} means that all concrete transition sequences are safely approximated for each thread individually – this solves the problem discussed in 1 in the previous section.

But then it must be that for each final state configuration in the concrete collecting semantics, ABSEXE (whenever it terminates) derives either an over-approximating final state configuration, or a timed-out configuration. Likewise, it must be that for each deadlocked configuration in the concrete collecting semantics, ABSEXE (whenever it terminates) derives either a deadlocked configuration or a timed-out configuration. □

All the details and the complete soundness proof of the presented algorithm are given in [16]. Note that Theorem 1 does not state that Algorithm 1 terminates for all possible inputs. This is because it might not terminate for some inputs – this problem is inherent in abstract execution.

Since abstract execution is not based on fixed point calculation of the collecting semantics in the traditional sense, widening and narrowing [17] cannot be used to alleviate this issue. Instead, timeouts on execution times and/or the

$$\begin{tikzcd}
c_0 \arrow[r, "prg"'] \arrow[d, shift right, "\alpha_{conf}"'] & c_1 \arrow[r, "prg"'] & c_2 \arrow[r, "prg"'] & \cdots \arrow[r, "prg"'] & c_q \arrow[d, shift right, "\alpha_{conf}"'] \\
\tilde{c}_0 \arrow[r, "\sim", "prg"'] \arrow[u, shift right, "\gamma_{conf}"'] & \tilde{c}_1 \arrow[r, "\sim", "prg"'] & \tilde{c}_2 \arrow[r, "\sim", "prg"'] & \cdots \arrow[r, "\sim", "prg"'] & \tilde{c}_w \arrow[u, shift right, "\gamma_{conf}"']
\end{tikzcd}$$

Fig. 4. Relation between concrete and abstract transitions

number of transitions can be set in different ways to guarantee termination of the analysis for all cases. This is further discussed in Sect. 8.

It should also be noted that, although this paper focuses on timing analysis, the defined algorithm could also be used for deadlock analyses and termination analyses [16]. If the returned sets are such that $\tilde{C}^d = \emptyset$ and $\tilde{C}^t = \emptyset$, then the analyzed program is free of deadlocks and always terminates for all initial states given by the configurations in $\tilde{C}$. Safe timing bounds for the program are then easily extracted from the configurations in $\tilde{C}^f$. On the other hand, if $\tilde{C}^d \neq \emptyset$, then the program might deadlock for the given initial states, and if $\tilde{C}^t \neq \emptyset$, then it might be that the program does not terminate also due to other reasons, such as an infinite loop in some thread. However, deadlock and/or termination analysis is not the main focus of the presented approach and many other more specialized techniques targeting these areas exist.

7 Example Analysis

To clarify and explain Algorithm 1, this section instantiates it for an example PPL program containing a parallel loop. The example shows how communication through shared memory and synchronization on locks are handled.

The purpose of the program in Fig. 5 is to increment the value of the variable x with $\sum_{i=1}^{4}(2i+3)$. The task of calculating the sum is equally divided onto two threads, T_1 and T_2. By definition, $\mathrm{Thrd} = \{\mathrm{T}_1, \mathrm{T}_2\}$, $\mathrm{Reg}_{\mathrm{T}_1} = \{\mathtt{p}, \mathtt{r}\}$, $\mathrm{Reg}_{\mathrm{T}_2} = \{\mathtt{p}, \mathtt{r}\}$, $\mathrm{Var} = \{\mathtt{x}\}$ and $\mathrm{Lck} = \{\mathtt{l}\}$. Note that p (and r) represents local memory within each thread; i.e. the register-name p (and r) can refer to two different memory locations – what location it refers to depends on which thread is considered. It is easy to see that x is a global variable when $\mathrm{Thrd}_{\tilde{c}} = \{\mathrm{T}_1, \mathrm{T}_2\}$ and that there are no global variables when $\mathrm{Thrd}_{\tilde{c}} = \{\mathrm{T}_1\}$ or $\mathrm{Thrd}_{\tilde{c}} = \{\mathrm{T}_2\}$.

$$
\begin{aligned}
\mathrm{T}_1 = (1, &[\mathtt{p} := \mathtt{p+1}]^1; [\mathtt{r} := \mathtt{r+2*p+3}]^2; [\mathtt{if\ p<2\ goto\ 1}]^3; [\mathtt{lock\ l}]^4;\\
&[\mathtt{load\ p\ from\ x}]^5; [\mathtt{p} := \mathtt{p+r}]^6; [\mathtt{store\ p\ to\ x}]^7; [\mathtt{unlock\ l}]^8; [\mathtt{halt}]^9)\\
\mathrm{T}_2 = (2, &[\mathtt{p} := \mathtt{p+1}]^1; [\mathtt{r} := \mathtt{r+2*p+3}]^2; [\mathtt{if\ p<4\ goto\ 1}]^3; [\mathtt{lock\ l}]^4;\\
&[\mathtt{load\ p\ from\ x}]^5; [\mathtt{p} := \mathtt{p+r}]^6; [\mathtt{store\ p\ to\ x}]^7; [\mathtt{unlock\ l}]^8; [\mathtt{halt}]^9)
\end{aligned}
$$

Fig. 5. Example: Program

For the sake of simplicity, the timing model (i.e. ABSTIME) as described in Table 1 gives that each statement within a thread has constant timing bounds; a '$-$' indicates that the entry is not applicable.

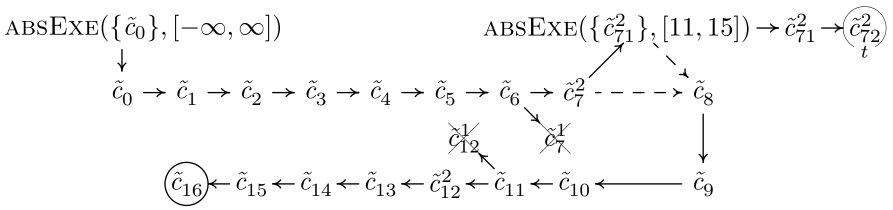
Assume that $\tilde{c}_0 = \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$ is as described in Table 2. Note that p and r for T_1, and r for T_2, are initialized to $[0, 0]$, and that p for T_2 is initialized to $[2, 2]$. Table 2 also collects all the configurations derived

Table 1. Example: Timing model

pc_{T} ($\mathrm{T} \in \mathrm{Thrd}$) :	1	2	3	4	5	6	7	8	9
$\textsc{absTime}(\tilde{c}, \mathrm{T}_1)$:	$[2,2]$	$[1,1]$	$[1,2]$	$[1,2]$	$[2,3]$	$[1,1]$	$[2,3]$	$[2,3]$	$-$
$\textsc{absTime}(\tilde{c}, \mathrm{T}_2)$:	$[2,2]$	$[1,1]$	$[4,5]$	$[5,6]$	$[2,5]$	$[2,2]$	$[2,4]$	$[2,3]$	$-$

by $\textsc{absExe}(\{\tilde{c}_0\}, [-\infty, \infty])$. A '$-$' indicates that the entry is not included in the configuration. Due to space limitations, the details on how $\tilde{t}^a_{\mathrm{T}_1}$ and $\tilde{t}^a_{\mathrm{T}_2}$ are calculated on each transition cannot be fully presented; please refer to [16] in case of unclarities. If a thread, $\mathrm{T} \in \mathrm{Thrd}$, is not included in Thrd_{exe} (as defined in Fig. 3), then $\tilde{t}^a_{\mathrm{T}}$ in $\tilde{c}_i$ is equal to $\tilde{t}^a_{\mathrm{T}}$ in $\tilde{c}_{i-1}$, where $i > 0$. If T is included in Thrd_{exe}, then $\tilde{t}^a_{\mathrm{T}}$ in $\tilde{c}_i$ is equal to $\tilde{t}^a_{\mathrm{T}} \mathbin{\tilde{+}_t} \textsc{absTime}(\tilde{c}_{i-1}, \mathrm{T})$ in $\tilde{c}_{i-1}$, unless T has been frozen and must have its accumulated execution time adapted to approximate the concrete spin-waiting.

Figure 6 shows the relation between the derived configurations. In the figure, final configurations are circled, timed-out configurations are circled and marked 't', and discontinued (invalid) configurations are crossed out. $\tilde{c}^1_7$ is discontinued since the timing constraints given by $\tilde{t}^a_{\mathrm{T}_2} \mathbin{\tilde{+}_t} \textsc{absTime}(\tilde{c}^1_7, \mathrm{T}_2) = [10, 11] \mathbin{\tilde{+}_t} [4, 5] = [14, 16]$ and the lock owner assignment deadline, $[-\infty, 12]$, give that T_2 cannot acquire `l` before T_1. $\tilde{c}^1_{12}$ is discontinued since T_1 cannot acquire `l` after reaching a `halt`-statement. Due to space limitations, the algorithm for calculating the deadline for the lock owner assignments made in the transitions from $\tilde{c}_6$ and $\tilde{c}_{11}$ cannot be presented; please refer to [16] in case of unclarities. Given $\tilde{c}^2_7$, a `store` to `x` in T_2 could affect the value loaded by T_1; however, the value loaded by T_1 cannot be affected after $\tilde{t}^a_{\mathrm{T}_1} \mathbin{\tilde{+}_t} \textsc{absTime}(\tilde{c}^2_7, \mathrm{T}_1) = [9, 12] \mathbin{\tilde{+}_t} [2, 3] = [11, 15]$, which is hence the timeout value for the recursive instance of $\textsc{absExe}$.



$$\begin{tikzcd}
\textsc{absExe}(\{\tilde{c}_0\}, [-\infty, \infty]) \arrow[d] & & & & & & & & \textsc{absExe}(\{\tilde{c}^2_{71}\}, [11, 15]) \arrow[r] \arrow[dr, dashed] & \tilde{c}^2_{71} \arrow[r] & \overset{}{\underset{t}{\enclose{circle}{\tilde{c}^2_{72}}}} \\
\tilde{c}_0 \arrow[r] & \tilde{c}_1 \arrow[r] & \tilde{c}_2 \arrow[r] & \tilde{c}_3 \arrow[r] & \tilde{c}_4 \arrow[r] & \tilde{c}_5 \arrow[r] & \tilde{c}_6 \arrow[r] \arrow[d] & \tilde{c}^2_7 \arrow[ur] \arrow[rr, dashed] & & \tilde{c}_8 \arrow[dd] & \\
 & & & & & \cancel{\tilde{c}^1_{12}} & \cancel{\tilde{c}^1_7} & & & & \\
 & \enclose{circle}{\tilde{c}_{16}} & \tilde{c}_{15} \arrow[l] & \tilde{c}_{14} \arrow[l] & \tilde{c}_{13} \arrow[l] & \tilde{c}^2_{12} \arrow[l] & \tilde{c}_{11} \arrow[l] \arrow[ul] & \tilde{c}_{10} \arrow[l] & & \tilde{c}_9 \arrow[ll] &
\end{tikzcd}$$

Fig. 6. Example: Configuration relations

It is apparent that $\textsc{absExe}(\{\tilde{c}_0\}, [-\infty, \infty]) = (\{\tilde{c}_{16}\}, \emptyset, \emptyset)$; i.e. $\tilde{c}_{16}$ is a final-state configuration and there are no deadlocked or timed-out configurations. It is thus easy to see that the program always terminates and that the estimated timing bounds are (Definition 2):

Table 2. Example: Derived configurations

$\tilde{c}$	pc_{T_1}	pc_{T_2}	$\tilde{\mathtt{r}}_{\mathrm{T}_1}\ \mathtt{p}$	$\tilde{\mathtt{r}}_{\mathrm{T}_1}\ \mathtt{r}$	$\tilde{\mathtt{r}}_{\mathrm{T}_2}\ \mathtt{p}$	$\tilde{\mathtt{r}}_{\mathrm{T}_2}\ \mathtt{r}$	$\tilde{t}^a_{\mathrm{T}_1}$	$\tilde{t}^a_{\mathrm{T}_2}$	$(\tilde{\mathtt{x}}\ \mathtt{x})\ \mathrm{T}_1$	$(\tilde{\mathtt{x}}\ \mathtt{x})\ \mathrm{T}_2$	$\tilde{\mathtt{l}}\ \mathtt{l}$
$\tilde{c}_0$	1	1	$[0,0]$	$[0,0]$	$[2,2]$	$[0,0]$	$[0,0]$	$[0,0]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_1$	2	2	$[1,1]$	$[0,0]$	$[3,3]$	$[0,0]$	$[2,2]$	$[2,2]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_2$	3	3	$[1,1]$	$[5,5]$	$[3,3]$	$[9,9]$	$[3,3]$	$[3,3]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_3$	1	3	$[1,1]$	$[5,5]$	$[3,3]$	$[9,9]$	$[4,5]$	$[3,3]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_4$	2	1	$[2,2]$	$[5,5]$	$[3,3]$	$[9,9]$	$[6,7]$	$[7,8]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_5$	3	1	$[2,2]$	$[12,12]$	$[3,3]$	$[9,9]$	$[7,8]$	$[7,8]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_6$	4	2	$[2,2]$	$[12,12]$	$[4,4]$	$[9,9]$	$[8,10]$	$[9,10]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, \tilde{\bot}_t, \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_7^1$	4	3	$[2,2]$	$[12,12]$	$[4,4]$	$[20,20]$	$[8,10]$	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \mathrm{T}_2, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_7^2$	5	3	$[2,2]$	$[12,12]$	$[4,4]$	$[20,20]$	$[9,12]$	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_1, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_{71}^2$	–	3	–	–	$[4,4]$	$[20,20]$	–	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_1, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_{72}^2$	–	4	–	–	$[4,4]$	$[20,20]$	–	$[14,16]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_1, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_8$	6	3	$[0,0]$	$[12,12]$	$[4,4]$	$[20,20]$	$[11,15]$	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_1, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_9$	7	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[12,16]$	$[14,16]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_1, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_{10}$	8	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[14,19]$	$[14,16]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_1, [-\infty,12], \bot_{thrd}, \tilde{\bot}_t)$
$\tilde{c}_{11}$	9	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[16,22]$	$[14,16]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \bot_{thrd}, [-\infty,12], \mathrm{T}_1, [16,22])$
$\tilde{c}_{12}^1$	9	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[16,22]$	$[14,16]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free}, \mathrm{T}_1, [-\infty,28], \mathrm{T}_1, [16,22])$
$\tilde{c}_{12}^2$	9	5	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[16,22]$	$[19,28]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken}, \mathrm{T}_2, [-\infty,28], \mathrm{T}_1, [16,22])$
$\tilde{c}_{13}$	9	6	$[12,12]$	$[12,12]$	$[12,12]$	$[20,20]$	$[16,22]$	$[21,33]$	$\{([12,12],[14,19])\}$	$\{(\tilde{\bot}_{val}, \tilde{\bot}_t)\}$	$(\mathit{taken}, \mathrm{T}_2, [-\infty,28], \mathrm{T}_1, [16,22])$
$\tilde{c}_{14}$	9	7	$[12,12]$	$[12,12]$	$[32,32]$	$[20,20]$	$[16,22]$	$[23,35]$	$\{([12,12],[14,19])\}$	$\{(\tilde{\bot}_{val}, \tilde{\bot}_t)\}$	$(\mathit{taken}, \mathrm{T}_2, [-\infty,28], \mathrm{T}_1, [16,22])$
$\tilde{c}_{15}$	9	8	$[12,12]$	$[12,12]$	$[32,32]$	$[20,20]$	$[16,22]$	$[25,39]$	$\{([12,12],[14,19])\}$	$\{([32,32],[25,39])\}$	$(\mathit{taken}, \mathrm{T}_2, [-\infty,28], \mathrm{T}_1, [16,22])$
$\tilde{c}_{16}$	9	9	$[12,12]$	$[12,12]$	$[32,32]$	$[20,20]$	$[16,22]$	$[27,42]$	$\{([12,12],[14,19])\}$	$\{([32,32],[25,39])\}$	$(\mathit{free}, \bot_{thrd}, [-\infty,28], \mathrm{T}_2, [27,42])$

$$\begin{cases} \text{BCET} = \min(\{\max(\{\min(\gamma_t(\tilde{t}^a_{\text{T}})) \mid \text{T} \in \text{Thrd}\}) \mid \\ \qquad \langle [\text{T}, pc_{\text{T}}, \tilde{\mathbb{r}}_{\text{T}}, \tilde{t}^a_{\text{T}}]_{\text{T}\in\text{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \in \{\tilde{c}_{16}\}\}) = 27 \\ \text{WCET} = \max(\{\max(\{\max(\gamma_t(\tilde{t}^a_{\text{T}})) \mid \text{T} \in \text{Thrd}\}) \mid \\ \qquad \langle [\text{T}, pc_{\text{T}}, \tilde{\mathbb{r}}_{\text{T}}, \tilde{t}^a_{\text{T}}]_{\text{T}\in\text{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \in \{\tilde{c}_{16}\}\}) = 42 \end{cases}$$

8 Conclusions and Future Work

This paper has presented a parallel programming language, PPL, with shared memory and synchronization primitives acting on locks, and an algorithm that derives safe approximations of the BCET and WCET of PPL programs, given some sets of initial states and a timing model of the underlying architecture. The algorithm is based on abstract execution, which itself is based on abstract interpretation of the PPL semantics, which helps proving the soundness of the algorithm due to the existence of a Galois Connection between final concrete and abstract configurations.

The recursive definition of the algorithm means that several auxiliary states might have to be searched when some thread loads global data to make sure that the loaded value is a safe approximation of the corresponding concrete value(s). However, since this only happens for a limited amount of steps (until no thread can affect the loaded value anymore), it is expected that this will not have a huge impact on the complexity of the algorithm.

The over-approximate lock owner assignment could cause a lot of auxiliary configurations to be added to the worklist. However, this is necessary to cover all the concrete possibilities for in which orders the locks are taken by the threads. The discontinuation of cases that are guaranteed to never occur concretely both lowers the complexity and increases the precision of the algorithm, and also avoids it to deadlock (which otherwise could happen even though the analyzed program might be deadlock free [16]).

Future work includes implementing and evaluating the algorithm. This includes deriving a timing model for some more or less realistic architecture. The precision of the timing model is expected to have a great impact on the complexity of the analysis presented in this paper. Therefore, efforts will also be made to decrease the overall complexity of the algorithm. How large parallel programs that will be analyzable by the presented approach remains an open question until the implementation and evaluation have been performed. However, it is already obvious that well-written parallel programs (i.e. programs in which communication through shared memory and synchronization on locks is minimized while thread-local computations are maximized; c.f. the example presented in Sect. 7) will be less complex to analyze.

Future work also includes extending PPL with more statements and operations so that a real programming language can be modeled. One example is to include different addressing modes so that for example arrays can be introduced and operated on. Another example could be to introduce other synchronization primitives, e.g. barriers.

As previously mentioned, the risk of nontermination is inherent in abstract execution since the technique is basically a symbolic execution of the analyzed program. Detecting deadlocks partly solves this issue. Solving the issue completely can be done by setting a finite upper limit on the number of abstract transitions. If the limit is reached, the analysis could simply terminate and result in an infinite upper bound on the execution time. Other timeouts could also be set, e.g. as upper limits on the calculated execution times of the threads in the analyzed program or as an upper limit on the run (i.e. execution) time of the analysis itself. Note that terminating the analysis before all possible transition sequences have been fully evaluated (i.e. before a final configuration has been reached) must result in an infinite estimation of the upper limit on the execution time (i.e. on the WCET).

The path-explosion problem is still an open issue. In the sequential case of abstract execution, this is solved by merging states [19]. However, this technique is not expected to be very successful for the analysis presented in this paper since all the concrete parts of the system state (i.e. the threads' program counters, the lock states and owners, etc.) would have to be equal for the states to be merged. Defining a more approximate abstract lock state could resolve this issue. How to make this abstraction will be a challenge for not losing too much precision in the analysis.

Acknowledgment. The research presented in this paper was supported by the Swedish Foundation for Strategic Research (SSF) via the project RALF3[1].

References

1. OpenMP: OpenMP Application Program Interface, Version 3.0 (May 2008), `http://www.openmp.org/mp-documents/spec30.pdf`
2. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution time problem — overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 1–53 (2008)
3. Gustafsson, J.: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden (May 2000)
4. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proc. 27th IEEE Real-Time Systems Symposium (RTSS 2006), Rio de Janeiro, Brazil, pp. 57–66. IEEE Computer Society (December 2006)
5. Ermedahl, A., Gustafsson, J., Lisper, B.: Deriving WCET bounds by abstract execution. In: Healy, C. (ed.) Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011), Porto, Portugal (July 2011)

[1] `http://www.es.mdh.se/projects/295-RALF3___Software_for_Embedded_High_Performance_Architectures`

6. Shaw, A.C.: Reasoning about time in higher-order software. IEEE Transactions on Software Engineering 15, 737–750 (1989)
7. Huber, B., Schoeberl, M.: Comparison of implicit path enumeration and model checking based WCET analysis. In: Proc. 9th International Workshop on Worst-Case Execution Time Analysis, WCET 2009 (2009)
8. Metzner, A.: Why model checking can improve WCET analysis. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 334–347. Springer, Heidelberg (2004)
9. Gustavsson, A., Ermedahl, A., Lisper, B., Pettersson, P.: Towards WCET analysis of multicore architectures using UPPAAL. In: Lisper, B. (ed.) Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), Brussels, Belgium, OCG, pp. 103–113 (July 2010)
10. Lv, M., Guan, N., Yi, W., Deng, Q., Yu, G.: Efficient instruction cache analysis with model checking. In: Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), pp. 33–36 (2010); Work-in-Progress Session
11. Wu, L., Zhang, W.: Bounding worst-case execution time for multicore processors through model checking. In: Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), pp. 17–20 (April 2010); Work-in-Progress Session
12. Gustavsson, A., Gustafsson, J., Lisper, B.: Toward static timing analysis of parallel software. In: Vardanega, T. (ed.) Proc. 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012). OpenAccess Series in Informatics (OASIcs), vol. 23, pp. 38–47 (July 2012)
13. Mittermayr, R., Blieberger, J.: Timing analysis of concurrent programs. In: Proc. 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012), pp. 59–68 (2012)
14. Potop-Butucaru, D., Puaut, I.: Integrated Worst-Case Execution Time Estimation of Multicore Applications. In: Proc. 13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013)
15. Ozaktas, H., Rochange, C., Sainrat, P.: Automatic WCET Analysis of Real-Time Parallel Applications. In: Proc. 13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013)
16. Gustavsson, A.: Static Timing Analysis of Parallel Software Using Abstract Execution. Licentiate thesis, Mälardalen University (2014), `http://www.es.mdh.se/publications/3025-Static_Timing_Analysis_of_Parallel_Software_Using_Abstract_Execution`
17. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd edn. Springer (2005) ISBN 3-540-65410-0
18. 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 Symposium on Principles of Programming Languages, Los Angeles, pp. 238–252 (January 1977)
19. Gustafsson, J., Ermedahl, A.: Merging techniques for faster derivation of WCET flow information using abstract execution. In: Kirner, R. (ed.) Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008), Prague, Czech Republic (July 2008)

Doomsday Equilibria for Omega-Regular Games

Krishnendu Chatterjee[1,⋆], Laurent Doyen[2],
Emmanuel Filiot[3,⋆⋆], and Jean-François Raskin[3,⋆⋆⋆]

[1] IST Austria
[2] LSV, ENS Cachan & CNRS
[3] CS-Université Libre de Bruxelles – U.L.B

Abstract. Two-player games on graphs provide the theoretical framework for many important problems such as reactive synthesis. While the traditional study of two-player zero-sum games has been extended to multi-player games with several notions of equilibria, they are decidable only for perfect-information games, whereas several applications require imperfect-information games.

In this paper we propose a new notion of equilibria, called doomsday equilibria, which is a strategy profile such that all players satisfy their own objective, and if any coalition of players deviates and violates even one of the players objective, then the objective of every player is violated.

We present algorithms and complexity results for deciding the existence of doomsday equilibria for various classes of ω-regular objectives, both for imperfect-information games, and for perfect-information games. We provide optimal complexity bounds for imperfect-information games, and in most cases for perfect-information games.

1 Introduction

Two-player games on finite-state graphs with ω-regular objectives provide the framework to study many important problems in computer science [31,29,14]. One key application area is synthesis of reactive systems [5,30,28]. Traditionally, the reactive synthesis problem is reduced to two-player zero-sum games, where vertices of the graph represent states of the system, edges represent transitions, one player represents a component of the system to synthesize, and the other player represents the purely adversarial coalition of all the other components. Since the coalition is adversarial, the game is zero-sum, i.e., the objectives of the two players are complementary. Two-player zero-sum games have been studied in great depth in literature [22,14,17].

Instead of considering all the other components as purely adversarial, a more realistic model is to consider them as individual players each with their own objective, as in protocol synthesis where the rational behavior of the agents is to first satisfy their own objective in the protocol before trying to be adversarial to the other agents. Hence,

⋆ Supported by Austrian Science Fund (FWF) Grant No P23499-N23, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award.

⋆⋆ Supported by the Belgian National Fund for Scientific Research.

⋆⋆⋆ Supported by ERC Start grant (279499: inVEST).

inspired by recent applications in protocol synthesis, the model of multi-player games on graphs has become an active area of research in graph games and reactive synthesis [1,16,32]. In a multi-player setting, the games are not necessarily zero-sum (i.e., objectives are not necessarily conflicting) and the classical notion of rational behavior is formalized as Nash equilibria [25]. Nash equilibria perfectly capture the notion of rational behavior in the absence of external criteria, i.e., the players are concerned only about their own payoff (internal criteria), and they are indifferent to the payoff of the other players. In the setting of synthesis, the more appropriate notion is the adversarial external criteria, where the players are as harmful as possible to the other players without sabotaging with their own objectives. This has inspired the study of refinements of Nash equilibria, such as secure equilibria [10] (that captures the adversarial external criteria), rational synthesis [16], and led to several new logics where the non-zero-sum equilibria can be expressed [11,13,24,33,23]. The complexity of Nash equilibria [32], secure equilibria [10], rational synthesis [16], and of the new logics has been studied recently [11,13,24,33].

Along with the theoretical study of refinements of equilibria, applications have also been developed in the synthesis of protocols. In particular, the notion of secure equilibria has been useful in the synthesis of mutual-exclusion protocol [10], and of fair-exchange protocols [20,7] (a key protocol in the area of security for exchange of digital signatures). One major drawback that all the notions of equilibria suffer is that the basic decision questions related to them are decidable only in the setting of perfect-information games (in a perfect-information games the players perfectly know the state and history of the game, whereas in imperfect-information games each player has only a partial view of the state space of the game), and in the setting of multi-player imperfect-information games they are undecidable [28]. However, the model of imperfect-information games is very natural because every component of a system has private variables not accessible to other components, and recent works have demonstrated that imperfect-information games are required in synthesis of fair-exchange protocols [19]. In this paper, we provide the first decidable framework that can model them.

We propose a new notion of equilibria which we call *doomsday-threatening* equilibria (for short, doomsday equilibria). A doomsday equilibria is a strategy profile such that all players satisfy their own objective, and if any coalition of players deviates and violates even one of the players objective, then doomsday follows (every player objective is violated). Note that in contrast to other notions of equilibria, doomsday equilibria consider deviation by an arbitrary set of players, rather than individual players. Moreover, in case of two-player non-zero-sum games they coincide with the secure equilibria [10] where objectives of both players are satisfied.

Example 1. Let us consider the two trees of Fig. 1. They model the possible behaviors of two entities Alice and Bob that have the objective of exchanging messages: m_{AB} from Alice to Bob, and m_{BA} from Bob to Alice. Assume for the sake of illustration that m_{AB} models the transfer of property of an house from Alice to Bob, while m_{BA} models the payment of the price of the house from Bob to Alice.

Having that interpretation in mind, let us consider the left tree. On the one hand, Alice has as primary objective (internal criterion) to reach either state 2 or state 4, states in which she has obtained the money, and she has a slight preference for 2 as

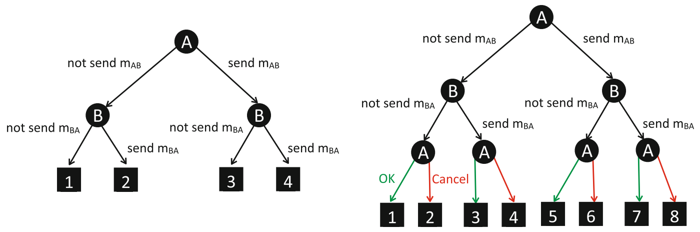


Fig. 1. A simple example in the domain of Fair Exchange Protocols

in that case she received the money while not transferring the property of her house to Bob, this corresponds to her adversarial external criterion. On the other hand, Bob would like to reach either state 3 or 4 (with again a slight preference for 3). Also, it should be clear that Alice would hate to reach 3 because she would have transferred the property of her house to Bob but without being paid. Similarly, Bob would hate to reach 2. To summarize, Alice has the following preference order on the final states of the protocol: $2 > 4 > 1 > 3$, while for Bob the order is $3 > 4 > 1 > 2$. Is there a *doomsday threatening equilibrium* in this game ? For such an equilibrium to exist, we must find a pair of strategies that please the two players for their primary objective (internal criterion): reach $\{2, 4\}$ for Alice and reach $\{3, 4\}$ for Bob. Clearly, this is only possible if at the root Alice plays "send m_{AB}", as otherwise we would not reach $\{1, 2\}$ violating the primary objective of Bob. But playing that action is not safe for Alice as Bob would then choose "not send m_{BA}" because he slightly prefers 3 to 4. It can be shown that the only rational way of playing (taking into account both internal and external criteria) is for Alice to play "not send m_{AB}" and for Bob would to play "not send m_{BA}". This way of playing is in fact the only secure equilibrium of the game but this is not what we hope from such a protocol.

The difficulty in this exchange of messages comes from the fact that Alice is starting the protocol by sending her part and this exposes her. To obtain a better behaving protocol, one solution is to add an additional stage after the exchanges of the two messages as depicted in the right tree of Fig. 1. In this new protocol, Alice has the possibility to cancel the exchange of messages (in practice this would be implemented by the intervention of a TTP[1]). For that new game, the preference orderings of the players are as follows: for Alice it is $3 > 7 > 1 = 2 = 4 = 6 = 8 > 5$, and for Bod it is $5 > 7 > 1 = 2 = 4 = 6 = 8 > 3$. Let us now show that there is a doomsday equilibrium in this new game. In the first round, Alice should play "send m_{AB}" as otherwise the internal objective of Bob would be violated, then Bob should play "send m_{BA}", and finally Alice should play "OK" to validate the exchange of messages. Clearly, this profile of strategies satisfies the first property of a doomsday equilibrium: both players have reached their primary objective. Second, let us show that no player has an incentive to deviate from that profile of strategies. First, if Alice deviates then Bob would play "not send m_{BA}", and we obtain a doomsday situation as both players have their primary objectives violated. Second, if Bob deviates by playing "not send m_{BA}", then

[1] TTP stands for *Trusted Third Party*.

Table 1. Summary of the results

objectives	safety	reachability	Büchi	co-Büchi	parity
perfect information	PSPACE-C	PTIME-C	PTIME-C	PTIME-C	PSPACE NP-HARD CONP-HARD
imperfect information	EXPTIME-C	EXPTIME-C	EXPTIME-C	EXPTIME-C	EXPTIME-C

Alice would cancel the protocol exchange which again produces a doomsday situation. So, no player has an incentive to deviate from the equilibrium and the outcome of the protocol is the desired one: the two messages have been fairly exchanged. So, we see that the threat of a doomsday that brings the action "Cancel" has a beneficial influence on the behavior of the two players. □

It should now be clear that multi-player games with doomsday equilibria provide a suitable framework to model various problems in protocol synthesis. In addition to the definition of doomsday equilibria, our main contributions are to present algorithms and complexity bounds for deciding the existence of such equilibria for various classes of ω-regular objectives both in the perfect-information and in the imperfect-information cases. In all cases but one, we establish the exact complexity. Our technical contributions are summarized in Table 1. More specifically:

1. *(Perfect-information games).* We show that deciding the existence of doomsday equilibria in multi-player perfect-information games is (i) PTIME-complete for reachability, Büchi, and coBüchi objectives; (ii) PSPACE-complete for safety objectives; and (iii) in PSPACE and both NP-hard and coNP-hard for parity objectives.
2. *(Imperfect-information games).* We show that deciding the existence of doomsday equilibria in multi-player imperfect-information games is EXPTIME-complete for reachability, safety, Büchi, coBüchi, and parity objectives.

The area of multi-player games and various notion of equilibria is an active area of research, but notions that lead to decidability in the imperfect-information setting and has applications in synthesis has largely been an unexplored area. Our work is a step towards it.

2 Doomsday Equilibria for Perfect Information Games

In this section, we define game arena with perfect information, ω-regular objectives, and doomsday equilibria.

Game Arena. An n-player game arena G with perfect information is defined as a tuple $(S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta)$ such that S is a nonempty finite set of *states*, $\mathcal{P} = \{S_1, S_2, \ldots, S_n\}$ is a partition of S into n classes of states, one for each player respectively, $s_{\mathsf{init}} \in S$ is the initial state, Σ is a finite set of actions, and $\Delta : S \times \Sigma \to S$ is the transition function.

Plays in n-player game arena G are constructed as follows. They start in the initial state s_{init}, and then an ω number of rounds are played as follows: the player that owns the current state s chooses a letter $\sigma \in \Sigma$ and the game evolves to the position $s' = \Delta(s, \sigma)$, then a new round starts from s'. So formally, a *play* in G is an infinite sequence $s_0 s_1 \dots s_n \dots$ such that (i) $s_0 = s_{\mathsf{init}}$ and (i) for all $i \geq 0$, there exists $\sigma \in \Sigma$ such that $s_{i+1} = \Delta(s_i, \sigma)$. The set of plays in G is denoted by $\mathsf{Plays}(G)$, and the set of finite prefixes of plays by $\mathsf{PrefPlays}(G)$. We denote by $\rho, \rho_1, \rho_i, \dots$ plays in G, by $\rho(0..j)$ the prefix of the play ρ up to position j and by $\rho(j)$ the position j in the play ρ. We also use π, π_1, π_2, ... to denote prefixes of plays. Let $i \in \{1, 2, \dots, n\}$, a prefix π belongs to Player i if $\mathsf{last}(\pi)$, the last state of π, belongs to Player i, i.e. $\mathsf{last}(\pi) \in S_i$. We denote by $\mathsf{PrefPlays}_i(G)$ the set of prefixes of plays in G that belongs to Player i.

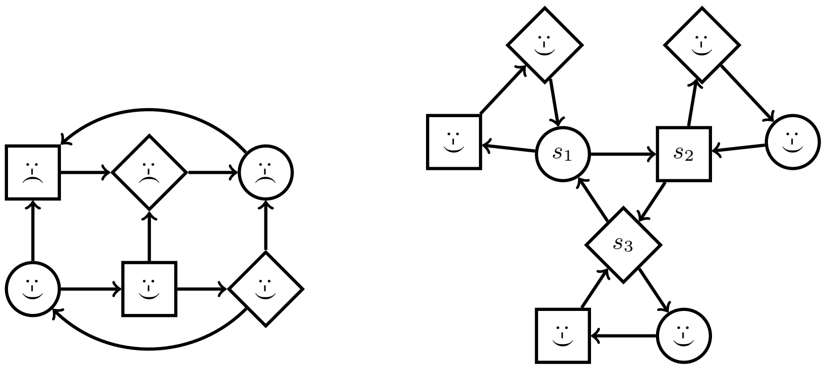
Strategies and Strategy Profiles. A *strategy* for Player i, for $i \in \{1, 2, \dots, n\}$, is a mapping $\lambda_i : \mathsf{PrefPlays}_i(G) \to \Sigma$ from prefixes of plays to actions. A *strategy profile* $\Lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$ is a tuple of strategies such that λ_i is a strategy of Player i. The strategy of Player i in Λ is denoted by Λ_i, and the the tuple of the remaining strategies $(\lambda_1, \dots, \lambda_{i-1}, \lambda_{i+1}, \dots, \lambda_n)$ by Λ_{-i}. For a strategy λ_i of Player i, we define its *outcome* as the set of plays that are consistent with λ_i: formally, $\mathsf{outcome}_i(\lambda_i)$ is the set of $\rho \in \mathsf{Plays}(G)$ such that for all $j \geq 0$, if $\rho(0..j) \in \mathsf{PrefPlays}_i(G)$, then $\rho(j+1) = \Delta(\rho(j), \lambda_i(\rho(0..j)))$. Similarly, we define the *outcome of a strategy profile* $\Lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$, as the unique play $\rho \in \mathsf{Plays}(G)$ such that for all positions j, for all $i \in \{1, 2, \dots, n\}$, if $\rho(j) \in \mathsf{PrefPlays}_i(G)$ then $\rho(j+1) = \Delta(\rho(j), \lambda_i(\rho(0..j)))$. Finally, given a state $s \in S$ of the game, we denote by G_s the game G whose initial state is replaced by s.

Winning Objectives. A *winning objective* (or an *objective* for short) φ_i for Player $i \in \{1, 2, \dots, n\}$ is a set of infinite sequences of states, i.e. $\varphi_i \subseteq S^\omega$. A strategy λ_i is *winning* for Player i (against all other players) w.r.t. an objective φ_i if $\mathsf{outcome}_i(\lambda_i) \subseteq \varphi_i$.

Given an infinite sequence of states $\rho \in S^\omega$, we denote by $\mathsf{visit}(\rho)$ the set of states that appear at least once along ρ, i.e. $\mathsf{visit}(\rho) = \{s \in S | \exists i \geq 0 \cdot \rho(i) = s\}$, and $\mathsf{inf}(\rho)$ the set of states that appear infinitely often along ρ, i.e. $\mathsf{inf}(\rho) = \{s \in S | \forall i \geq 0 \cdot \exists j \geq i \cdot \rho(i) = s\}$. We consider the following types of winning objectives:

– a *safety objective* is defined by a subset of states $T \subseteq S$ that has to be never left: $\mathsf{safe}(T) = \{\rho \in S^\omega \mid \mathsf{visit}(\rho) \subseteq T\}$;
– a *reachability objective* is defined by a subset of states $T \subseteq S$ that has to be reached: $\mathsf{reach}(T) = \{\rho \in S^\omega \mid \mathsf{visit}(\rho) \cap T \neq \emptyset\}$;
– a *Büchi objective* is defined by a subset of states $T \subseteq S$ that has to be visited infinitely often: $\mathsf{Büchi}(T) = \{\rho \in S^\omega \mid \mathsf{inf}(\rho) \cap T \neq \emptyset\}$;
– a *co-Büchi objective* is defined by a subset of states $T \subseteq S$ that has to be reached eventually and never be left: $\mathsf{coBüchi}(T) = \{\rho \in S^\omega \mid \mathsf{inf}(\rho) \subseteq T\}$;
– let $d \in \mathbb{N}$, a *parity objective with d priorities* is defined by a priority function $p : S \to \{0, 1, \dots, d\}$ as the set of plays such that the smallest priority visited infinitely often is even: $\mathsf{parity}(p) = \{\rho \in S^\omega | \min\{p(s) \mid s \in \mathsf{inf}(\rho)\}$ is even$\}$.

Büchi, co-Büchi and parity objectives φ are called *tail objectives* because they enjoy the following closure property: for all $\rho \in \varphi$ and all $\pi \in S^*$, $\rho \in \varphi$ iff $\pi \cdot \rho \in \varphi$.



(a) Doomsday (Safety) (b) Büchi objectives

Fig. 2. Examples of doomsday equilibria for Safety and Büchi objectives

Finally, given an objective $\varphi \subseteq S^\omega$ and a subset $P \subseteq \{1, \ldots, n\}$, we write $\langle\langle P \rangle\rangle \varphi$ to denote the set of states s from which the players from P can cooperate to enforce φ when they start playing in s. Formally, $\langle\langle P \rangle\rangle \varphi$ is the set of states s such that there exists a set of strategies $\{\lambda_i \mid i \in P\}$ in G_s, one for each player in P, such that $\bigcap_{i \in P} \mathsf{outcome}_i(\lambda_i) \subseteq \varphi$.

Doomsday Equilibria. A strategy profile $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ is a *doomsday-threatening equilibrium* (doomsday equilibrium or DE for short) if:

1. it is winning for all the players, i.e. $\mathsf{outcome}(\Lambda) \in \bigcap_i \varphi_i$;
2. each player is able to retaliate in case of deviation: for all $1 \leq i \leq n$, for all $\rho \in \mathsf{outcome}_i(\lambda_i)$, if $\rho \notin \varphi_i$, then $\rho \in \bigcap_{j=1}^{j=n} \overline{\varphi_j}$ (doomsday), where $\overline{\varphi_j}$ denotes the complement of φ_j in S^ω.

In other words, when all players stick to their strategies then they all win, and if any arbitrary coalition of players deviates and makes even just one other player lose then this player retaliates and ensures a doomsday, i.e. all players lose.

Relation with Secure Equilibria In two-player games, the doomsday equilibria coincide with the notion of secure equilibrium [10] where both players satisfy their objectives. In secure equilibria, for all $i \in \{1, 2\}$, any deviation of Player i that does not decrease her payoff does not decrease the payoff of Player $3-i$ either. In other words, if a deviation of Player i decreases (strictly) the payoff of Player $3-i$, i.e. φ_{3-i} is not satisfied, then it also decreases her own payoff, i.e. φ_i is not satisfied. A two-player secure equilibrium where both players satisfy their objectives is therefore a doomsday equilibrium.

Example 2. Fig. 2 gives two examples of games with safety and Büchi objectives respectively. Actions are in bijection with edges so they are not represented.

(Safety) Consider the 3-player game arena with perfect information of Fig. 2(a) and safety objectives. Unsafe states for each player are given by the respective nodes of the upper part. Assume that the initial state is one of the safe states. This example models a situation where three countries are in peace until one of the countries, say country i, decides to attack country j. This attack will then necessarily be followed by a doomsday situation: country j has a strategy to punish all other countries. The doomsday equilibrium in this example is to play safe for all players.

(Büchi) Consider the 3-player game arena with perfect information of Fig. 2(b) with Büchi objectives for each player: Player i wants to visit infinitely often one of its "happy" states. The position of the initial state does not matter. To make things more concrete, let us use this game to model a protocol where 3 players want to share in each round a piece of information made of three parts: for all $i \in \{1, 2, 3\}$, Player i knows information $i \; mod \; 3 + 1$ and $i \; mod \; 3 + 2$. Player i can send or not these informations to the other players. This is modeled by the fact that Player i can decide to visit the happy states of the other players, or move directly to $s_{(i \; mod \; 3)+1}$. The objective of each player is to have an infinite number of successful rounds where they get all information.

There are several doomsday equilibria. As a first one, let us consider the situation where for all $i \in \{1, 2, 3\}$, if Player i is in state s_i, first it visits the happy states, and when the play comes back in s_i, it moves to $s_{(i \; mod \; 3)+1}$. This defines an infinite play that visits all the states infinitely often. Whenever some player deviates from this play, the other players retaliate by always choosing in the future to go to the next s state instead of taking their respective loops. Clearly, if all players follow their respective strategies all happy states are visited infinitely often. Now consider the strategy of Player i against two strategies of the other players that makes him lose. Clearly, the only way Player i loses is when the two other players eventually never take their states, but then all the players lose.

As a second one, consider the strategies where Player 2 and Player 3 always take their loops but Player 1 never takes his loop, and such that whenever the play deviates, Player 2 and 3 retialate by never taking their loops. For the same reasons as before this strategy profile is a doomsday equilibrium.

Note that the first equilibrium requires one bit of memory for each player, to remember if they visit their s state for the first or second times. In the second equilibrium, only Player 2 and 3 needs a bit of memory. An exhaustive analysis shows that there is no memoryless doosmday equilibrium in this example. □

3 Complexity of DE for Perfect Information Games

In this section, we prove the following results:

Theorem 1. *The problem of deciding the existence of a doomsday equilibrium in an n-player perfect information game arena and n objectives $(\varphi_i)_{1 \leq i \leq n}$ is:*

- PTIME-C *if the objectives $(\varphi_i)_{1 \leq i \leq n}$ are either all Büchi, all co-Büchi or all reachability objectives, hardness already holds for 2-player game arenas,*
- NP-HARD, CONP-HARD *and in* PSPACE *if $(\varphi_i)_{1 \leq i \leq n}$ are parity objectives, hardness already holds for 2-player game arenas,*
- PSPACE-C *if $(\varphi_i)_{1 \leq i \leq n}$ are safety objectives, and* PTIME-C *for game arenas with a fixed number of players.*

In the sequel, game arena with perfect information are just called game arena.

Tail Objectives. We first present a generic algorithm that works for any tail objective and then analyze its complexity for the different cases. Then we establish the lower bounds. Let us consider the following algorithm:

- compute the retaliation region of each player: $R_i = \langle\langle i \rangle\rangle(\varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j})$;
- check for the existence of a play within $\bigcap_{i=1}^{i=n} R_i$ that satisfies all the objectives φ_i.

The correctness of this generic procedure is formalized in the following lemma:

Lemma 1. *Let $G = (S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta)$ be an n-player game arena with n tail objectives $(\varphi_i)_{1 \le i \le n}$. Let $R_i = \langle\langle i \rangle\rangle(\varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j})$ be the retaliation region for Player i. There is a doomsday equilibrium in G iff there exists an infinite play that* (1) *belongs to $\bigcap_{i=1}^{i=n} \varphi_i$ and* (2) *stays within the set of states $\bigcap_{i=1}^{i=n} R_i$.*

Proof. First, assume that there exists an infinite play ρ such that $\rho \in \bigcap_i (\varphi_i \cap R_i^{\omega})$. From ρ, and the retaliating strategies that exist in all states of R_i for each player, we show the existence of DE $\Lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$. Player i plays strategy λ_i as follows: he plays according to the choices made in ρ as long as all the other players do so, and as soon as the play deviates from ρ, Player i plays his retaliating strategy (when it is his turn to play).

First, let us show that if Player j, for some $j \neq i$, deviates and the turn comes back to Player i in a state s then $s \in R_i$. Assume that Player j deviates when he is in some $s' \in S_j$. As before there was no deviation, by definition of ρ, s' belongs to R_i. But no matter what the adversary are doing in a state that belongs to R_i, the next state must be a state that belongs to R_i (there is only the possibility to leave R_i when Player i plays). So, by induction on the length of the segment of play that separates s' and s, we can conclude that s belongs to R_i. From s, Player i plays a retaliating strategy and so all the outcomes from s are in $\varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$, and since the objective are tails, the prefix up to s is not important and we get (from s_{init}) $\mathsf{outcome}_i(\lambda_i) \subseteq \varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$. Therefore the second property of the definition of doomsday equilibria is satisfied. Hence Λ is a DE.

Let us now consider the other direction. Assume that Λ is a DE. Then let us show that $\rho = \mathsf{outcome}(\Lambda)$ satisfies properties (1) and (2). By definition of DE, we know that ρ is winning for all the players, so (1) is satisfied. Again by definition of DE, $\mathsf{outcome}(\Lambda_i) \subseteq \varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$. Let s be a state of ρ and π the prefix of ρ up to s. For all outcomes ρ' of Λ_i in G_s, we have $\pi\rho' \in \varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$, and since the objectives are tail, we get $\rho' \in \varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$. Hence $s \in R_i$. Since this property holds for all i, we get $s \in \bigcap_i R_i$, and (2) is satisfied. □

Accordingly, we obtain the following upper-bounds:

Lemma 2. *The problem of deciding the existence of a doomsday equilibrium in an n-player game arena can be decided in* PTIME *for Büchi and co-Büchi objectives, and in* PSPACE *for parity objectives.*

Proof. By Lemma 1 one first needs to compute the retaliation regions R_i for all $i \in \{1, \dots, n\}$. Once the sets R_i have been computed, it is clear that the existence of a play winning for all players is decidable in PTIME for all the three types of objectives. For the Büchi and the co-Büchi cases, let us show how to compute the retaliation regions R_i. We start with Büchi and we assume that each player wants to visit a set of states T_i infinitely often. Computing the sets R_i boils down to computing the set of states s from which Player i has a strategy to enforce the objective (in LTL syntax) $\Box\Diamond T_i \vee \bigwedge_{j=1}^{j=n} \Diamond\Box\overline{T_j}$, which is equivalent to the formula $\Box\Diamond T_i \vee \Diamond\Box \bigcap_{j=1}^{j=n} \overline{T_j}$. This is equivalent to a disjunction of a Büchi and a co-Büchi objective, which is thus equivalent to a Streett objective with one Streett pair and can be solved in PTime with a classical algorithm, e.g. [27]. Similarly, for co-Büchi objectives, one can reduce the computation of the regions R_i in polynomial time to the disjunction of a Büchi objective and a co-Büchi objective.

For the parity case, the winning objectives for the retaliation sets can be encoded compactly as Muller objectives defined by a propositional formula using one proposition per state. Then they can be solved in PSPACE using the algorithm of Emerson and Lei presented in [15]. $\square$

Let us now establish the lower bounds.

Lemma 3. *The problem of deciding the existence of a DE in an n-player game arena is* PTIME-HARD *for Büchi and co-Büchi objectives,* NP-HARD *and* CONP-HARD *for parity objectives. All the hardness results hold even for 2-player game arenas.*

Proof. The hardness for Büchi and co-Büchi objectives holds already for 2 players. We describe the reduction for Büchi and it is similar for co-Büchi. We reduce the problem of deciding the winner in a two-player zero-sum game arena G with a Büchi objective (known as a PTIME-HARD problem [18]) to the existence of a DE for Büchi objectives with two players. Consider a copy G' of the game arena G and the following two objectives: Player 1 has the same Büchi objective as Player 1 in G, and Player 2 has a trivial Büchi objective (i.e. all states are Büchi states). Then clearly there exists a DE in G' iff Player 1 has a winning strategy in G. Details are given in the long version of this paper [8].

For parity games, we can reduce zero-sum two-player games with a conjunction of parity objectives (known to be CONP-HARD [12]) to the existence of a DE in a three player game with parity objectives. Similarly, we can reduce the problem of deciding the winner in a two-player zero-sum game with a disjunction of parity objectives (known to be NP-HARD [12]) to the existence of a DE in a two-player game with parity objectives. The main idea in the two cases is to construct a game arena where one of the players can retaliate iff Player 1 in the original two-player zero-sum game has a winning strategy. Details are given in the long version of this paper [8]. $\square$

As a corollary of this result, deciding the existence of a secure equilibrium in a 2-player game such that both players satisfy their parity objectives is NP-HARD.

Reachability Objectives. We now establish the complexity of deciding the existence of a doomsday equilibria in an n-player game with reachability objectives. We first establish an important property for reachability objectives:

Proposition 1. *Let $G = (S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta)$ be a game arena, and $(T_i)_{1 \leq i \leq n}$ be n subsets of S. Let Λ be a doomsday equilibrium in G for the reachability objectives $(\mathsf{Reach}(T_i))_{1 \leq i \leq n}$. Let s the first state in $\mathsf{outcome}(\Lambda)$ such that $s \in \bigcup_i T_i$. Then every player has a strategy from s, against all the other players, to reach his target set.*

Proof. W.l.o.g. we can assume that $s \in T_1$. If some player, say Player 2, as no strategy from s to reach his target set T_2, then necessarily $s \notin T_2$ and by determinancy the other players have a strategy from s to make Player 2 lose. This contradicts the fact that Λ is a doomsday equilibrium as it means that Λ_2 is not a retaliating strategy. □

Lemma 4. *The problem of deciding the existence of a doomsday equilibrium in an n-player game with reachability objectives is in* PTIME.

Proof. The algorithm consists in:

(1) computing the sets R_i from which player i can retaliate, i.e. the set of states s from which Player i has a strategy to force, against all other players, an outcome such that $\Diamond T_i \vee (\bigwedge_{j=1}^{j=n} \Box \overline{T_j})$. This set can be obtained by first computing the set of states $\langle\langle i \rangle\rangle \Diamond T_i$ from which Player i can force to reach T_i. It is done in PTIME by solving a classical two-player reachability game. Then the set of states where Player i has a strategy λ_i such that $\mathsf{outcome}_i(\lambda_i) \models \Box((\bigcap_{j=1}^{j=n} \overline{T_j}) \vee \langle\langle i \rangle\rangle \Diamond T_i)\}$, that is to confine the plays in states that do not satisfy the reachability objectives of the adversaries or from where Player i can force its own reachability objective. Again this can be done in PTIME by solving a classical two-player safety game.

(2) then, checking the existence of some $i \in \{1, \ldots, n\}$ and some finite path π starting from s_{init} and that stays within $\bigcap_{j=1}^{j=n} R_j$ before reaching a state s such that $s \in T_i$ and $s \in \bigcap_{j=1}^{j=n} \langle\langle j \rangle\rangle \Diamond T_j$.

Let us now prove the correctness of our algorithm. From its output, we can construct the strategy profile Λ where each Λ_j ($j = 1, \ldots, n$) is as follows: follow π up to the point where either another player deviates and then play the retaliating strategy available in R_i, or to the point where s is visited for the first time and then play according to a strategy (from s) that force a visit to T_i no matter how the other players are playing. Clearly, Λ witnesses a DE. Indeed, if s is reached, then all players have a strategy to reach their target set (including Player i since $s \in T_i$) . By playing so they will all eventually reach it. Before reaching s, if some of them deviate, the other have a strategy to retaliate as π stays in $\bigcap_{j=1}^{j=n} R_j$. The other direction follows from Proposition 1. □

Lemma 5. *The problem of deciding the existence of a DE in a 2-player game with reachability objectives is* PTIME-HARD.

Proof. It is proved by an easy reduction from the And-Or graph reachability problem [18]: if reachability is trivial for one of the two players, the existence of a doomsday equilibrium is equivalent to the existence of a winning strategy for the other player in a two-player zero sum reachability game. □

Safety Objectives. We establish the complexity of deciding the existence of a doomsday equilibrium in an n-player game with perfect information and safety objectives.

Lemma 6 (EASYNESS). *The existence of a doomsday equilibrium in an n-player game with safety objectives can be decided in* PSPACE, *and in* PTIME *for game arenas with a fixed number of players.*

Proof. We start with the general case where the number of players is not fixed and is part of the input. Let us consider an n-player game arena $G = (S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta)$ and n safety objectives $\mathsf{safe}(T_1), \dots, \mathsf{safe}(T_n)$ for $T_1 \subseteq S, \dots, T_n \subseteq S$. The algorithm is composed of the following two steps:

(1) For each Player i, compute the set of states $s \in S$ in the game such that Player i can retaliate whenever necessary, i.e. the set of states s from where there exists a strategy λ_i for Player i such that $\mathsf{outcome}_i(\lambda_i)$ satisfies $\neg(\Box T_i) \rightarrow \bigwedge_{j=1}^{j=n} \neg\Box T_j$, or equivalently $\neg(\Diamond\overline{T_i}) \vee \bigwedge_{j=1}^{j=n} \Diamond\overline{T_j}$. This can be done in PSPACE using a result by Alur et al. (Theorem 5.4 of [2]) on solving two-player games whose Player 1's objective is defined by Boolean combinations of LTL formulas that use only $\Diamond$ and $\wedge$. We denote by R_i the set of states in G where Player i has a strategy to retaliate.

(2) then, verify whether there exists an infinite path in $\bigcap_{i=1}^{i=n}(\mathsf{safe}(T_i) \cap R_i)$.

Now, let us establish the correctness of this algorithm. Assume that an infinite path exists in $\bigcap_{i=1}^{i=n}(\mathsf{safe}(T_i) \cap R_i)$. The strategies λ_i for each Player i are defined as follows: play the moves that are prescribed as long as every other players do so, and as soon as the play deviates from the infinite path, play the retaliating strategy.

It is easy to see that the profile of strategies $\Lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$ is a DE. Indeed, the states are all safe for all players as long as they play their strategies. Moreover, as before deviation the play is within $\bigcap_{i=1}^{i=n} R_i$, if Player j deviates, we know that the state that is reached after deviation is still in $\bigcap_{j=1}^{j=n} R_j$ and therefore the other players can retaliate.

Second, assume that $\Lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$ is a DE in the n-player game G for the safety objectives $(\mathsf{safe}(T_i))_{1 \le i \le n}$. Let $\rho = \mathsf{outcome}(\lambda_1, \lambda_2, \dots, \lambda_n)$. By definition of doomsday equilibrium, we know that all states appearing in ρ satisfy all the safety objectives, i.e. $\rho \models \bigwedge_{i=1}^{i=n} \Box T_i$. Let us show that the play also remains within $\bigcap_{i=1}^{i=n} R_i$. Let s be a state of ρ, $i \in \{1, \dots, n\}$, and π the finite prefix of ρ up to s. By definition of DE we have $\mathsf{outcome}(\lambda_i) \models \Box T_i \vee \bigwedge_{j=1}^{j=n} \Diamond\overline{T_j}$. Therefore for all outcomes ρ' of λ_i in G_s, $\pi\rho' \models \Box T_i \vee \bigwedge_{j=1}^{j=n} \Diamond\overline{T_j}$. Moreover, $\pi \models \bigwedge_{j=1}^{j=n} \Box T_j$ since it is a prefix of ρ. Therefore $\rho' \models \Box T_i \vee \bigwedge_{j=1}^{j=n} \Diamond\overline{T_j}$ and $s \in R_i$. Since it holds for all $i \in \{1, \dots, n\}$, we get $s \in \bigcap_{i=1}^{i=n} R_i$.

Let us now turn to the case where the number of players is fixed. Then clearly, in the construction above, all the LTL formulas are of fixed size and so all the associated games can then be solved in polynomial time. □

Lemma 7 (HARDNESS). *The problem of deciding the existence of a doomsday equilibrium in an n-player game with safety objectives is* PSPACE-HARD, *and* PTIME-HARD *when the number of players is fixed.*

Proof. For the general case, we present a reduction from the problem of deciding the winner in a zero-sum two-player game with a conjunction of k reachability objectives (aka generalized reachability games), which is a PSPACE-C problem [3]. The idea of the reduction is to construct a non-zero sum $(k + 1)$-player game where one of the players

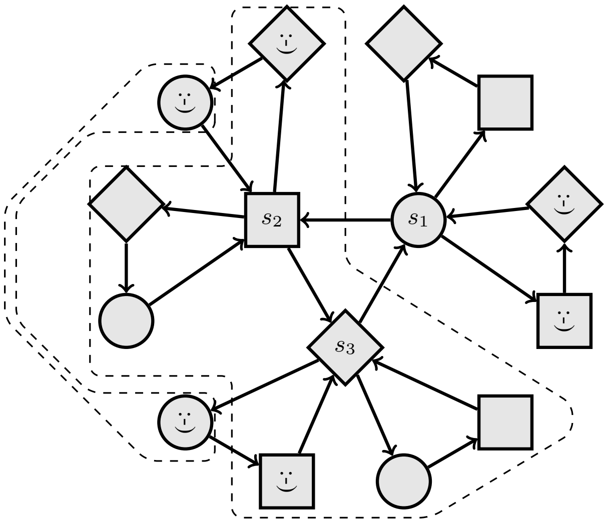


Fig. 3. Game arena with imperfect information and Büchi objectives. Only undistinguishable states of Player 1 (circle) are depicted. Observations are symmetric for the other players.

has a retaliating strategy iff there is a winning strategy in the generalized reachability game.

When the number of players is fixed, PTIME-HARDNESS is proved by an easy reduction from the And-Or graph reachability problem [18]. □

4 Complexity of DE for Imperfect Information Games

In this section, we define n-player game arenas with imperfect information. We adapt to this context the notions of observation, observation of a play, observation-based strategies, and we study the notion of doomsday equilibria when players are restricted to play observation-based strategies.

Game Arena with Imperfect Information. An n-player game arena with *imperfect information* is a tuple $G = (S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta, (O_i)_{1\leq i\leq n})$ such that $(S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta)$ is a game arena (of perfect information) and for all i, $1 \leq i \leq n$, $O_i \subseteq 2^S$ is a *partition* of S. Each block in O_i is called an *observation* of Player i. We assume that the players play in a predefined order[2]: for all $i \in \{1, \ldots, n\}$, all $q \in S_i$ and all $\sigma \in \Sigma$, $\Delta(q, \sigma) \in S_{(i \ mod \ n)+1}$.

Observations. For all $i \in \{1, \ldots, n\}$, we denote by $O_i(s) \subseteq S$ the block in O_i that contains s, that is the observation that Player i has when he is in state s. We say that two states s, s' are *undistinguishable* for Player i if $O_i(s) = O_i(s')$. This defines an

[2] This restriction is not necessary to obtain the results presented in this section (e.g. Theorem 2) but it makes some of our notations lighter.

equivalence relation on states that we denote by $\sim_i$. The notions of plays and prefixes of plays are slight variations from the perfect information setting: a play in G is a sequence $\rho = s_0, \sigma_0, s_1, \sigma_1, \dots \in (S \cdot \Sigma)^\omega$ such that $s_0 = s_{\mathsf{init}}$, and for all $j \geq 0$, we have $s_{j+1} = \Delta(s_j, \sigma_j)$. A prefix of play is a sequence $\pi = s_0, \sigma_0, s_1, \sigma_1, \dots, s_k \in (S \cdot \Sigma)^* \cdot S$ that can be extended into a play. As in the perfect information setting, we use the notations $\mathsf{Plays}(G)$ and $\mathsf{PrefPlays}(G)$ to denote the set of plays in G and its set of prefixes, and $\mathsf{PrefPlays}_i(G)$ for the set of prefixes that end in a state that belongs to Player i. While actions are introduced explicitly in our notion of play and prefix of play, their visibility is limited by the notion of observation. The *observation* of a play $\rho = s_0, \sigma_0, s_1, \sigma_1, \dots$ by Player i is the infinite sequence written $\mathsf{Obs}_i(\rho) \in (O_i \times (\Sigma \cup \{\tau\})^\omega$ such that for all $j \geq 0$, $\mathsf{Obs}_i(\rho)(j) = (O_i(s_j), \tau)$ if $s_j \notin S_i$, and $\mathsf{Obs}_i(\rho)(j) = (O_i(s_j), \sigma_j)$ if $s_j \in S_i$. Thus, only actions played by Player i are visible along the play, and the actions played by the other players are replaced by τ. The observation $\mathsf{Obs}_i(\pi)$ of a prefix π is defined similarly. Given an infinite sequence of observations $\eta \in (O_i \times (\Sigma \cup \{\tau\}))^\omega$ for Player i, we denote by $\gamma_i(\eta)$ the set of plays in G that are compatible with η, i.e. $\gamma_i(\eta) = \{\rho \in \mathsf{Plays}(G) \mid \mathsf{Obs}_i(\rho) = \eta\}$. The functions γ_i are extended to prefixes of sequences of observations naturally.

Observation-Based Strategies and Doomsday Equilibria. A strategy λ_i of Player i is *observation-based* if for all prefixes of plays $\pi_1, \pi_2 \in \mathsf{PrefPlays}_i(G)$ such that $\mathsf{Obs}_i(\pi_1) = \mathsf{Obs}_i(\pi_2)$, it holds that $\lambda_i(\pi_1) = \lambda_i(\pi_2)$, i.e. while playing with an observation-based strategy, Player i plays the same action after undistinguishable prefixes. A strategy profile Λ is observation-based if each Λ_i is observation-based. Winning objectives, strategy outcomes and winning strategies are defined as in the perfect information setting. We also define the notion of outcome relative to a prefix of a play. Given an observation-based strategy λ_i for Player i, and a prefix $\pi = s_0, \sigma_0, \dots, s_k \in \mathsf{PrefPlays}_i(G)$, the strategy λ_i^π is defined for all prefixes $\pi' \in \mathsf{PrefPlays}_i(G_{s_k})$ where G_{s_k} is the game arena G with initial state s_k, by $\lambda_i^\pi(\pi') = \lambda_i(\pi \cdot \pi')$. The set of outcomes of the strategy λ_i *relative to* π is defined by $\mathsf{outcome}_i(\pi, \lambda_i) = \pi \cdot \mathsf{outcome}_i(\lambda_i^\pi)$.

The notion of doomsday equilibrium is defined as for games with perfect information but with the additional requirements that *only* observation-based strategies can be used by the players. Given an n-player game arena with imperfect information G and n winning objectives $(\varphi_i)_{1 \leq i \leq n}$ (defined as in the perfect information setting), we want to solve the problem of deciding the existence of an *observation-based strategy profile* Λ which is a doomsday equilibrium in G for $(\varphi_i)_{1 \leq i \leq n}$.

Example 3. Fig. 3 depicts a variant of the example in the perfect information setting, with imperfect information. In this example let us describe the situation for Player 1. It is symmetric for the other players. Assume that when Player 2 or Player 3 send their information to Player 1 (modeled by a visit to his happy states), Player 1 cannot distinguish which of Player 2 or 3 has sent the information, e.g. because of the usage of a cryptographic primitive. Nevertheless, let us show that there exists doomsday equilibrium. Assume that the three players agree on the following protocol: Player 1 and 2 send their information but not Player 3.

Let us show that this sequence witnesses a doomsday equilibrium and argue that this is the case for Player 1. From the point of view of Player 1, if all players follow this profile of strategies then the outcome is winning for Player 1. Now, let us consider two types of deviation. First, assume that Player 2 does not send his information (i.e. does not visit the happy states). In that case Player 1 will observe the deviation and can retaliate by not sending his own information. Therefore all the players are losing. Second, assume that Player 2 does not send his information but Player 3 does. In this case it is easy to verify that Player 1 cannot observe the deviation and so according to his strategy will continue to send his information. This is not problematic because all the plays that are compatible with Player 1's observations are such that: (i) they are winning for Player 1 (note that it would be also acceptable that all the sequence are either winning for Player 1 or losing for all the other players), and (ii) Player 1 is always in position to retaliate along this sequence of observations. In our solution below these two properties are central and will be called *doomsday compatible* and *good for retaliation*. □

Generic Algorithm. We present a generic algorithm to test the existence of an observation-based doomsday equilibrium in a game of imperfect information. To present this solution, we need two additional notions: sequences of observations which are *doomsday compatible* and prefixes which are *good for retaliation*. These two notions are defined as follows. In a game arena $G = (S, \mathcal{P}, s_{\mathsf{init}}, \Sigma, \Delta, (O_i)_{1 \le i \le n})$ with imperfect information and winning objectives $(\varphi_i)_{1 \le i \le n}$,

- a sequence of observations $\eta \in (O_i \times (\Sigma \cup \{\tau\}))^\omega$ is *doomsday compatible* (for Player i) if $\gamma_i(\eta) \subseteq \varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$, i.e. all plays that are compatible with η are either winning for Player i, or not winning for any other player,
- a prefix $\kappa \in (O_i \times (\Sigma \cup \{\tau\}))^* \cdot O_i$ of a sequence of observations is *good for retaliation* (for Player i) if there exists an observation-based strategy λ_i^R such that for all prefixes $\pi \in \gamma_i(\kappa)$ compatible with κ, $\mathsf{outcome}(\pi, \lambda_i^R) \subseteq \varphi_i \cup \bigcap_{j=1}^{j=n} \overline{\varphi_j}$.

The next lemma shows that the notions of sequences of observations that are doomsday compatible and good for retaliation prefixes are important for studying the existence of doomsday equilibria for imperfect information games.

Lemma 8. *Let G be an n-player game arena with imperfect information and winning objectives φ_i, $1 \le i \le n$. There exists a doomsday equilibrium in G if and only if there exists a play ρ in G such that:*

(F_1) $\rho \in \bigcap_{i=1}^{i=n} \varphi_i$, *i.e. ρ is winning for all the players,*
(F_2) *for all Player i, $1 \le i \le n$, for all prefixes κ of* $\mathsf{Obs}_i(\rho)$, *κ is good for retaliation for Player i,*
(F_3) *for all Player i, $1 \le i \le n$,* $\mathsf{Obs}_i(\rho)$ *is doomsday compatible for Player i.*

Proof. First, assume that conditions (F_1), (F_2) and (F_3) hold and show that there exists a DE in G. We construct a DE $(\lambda_1, \ldots, \lambda_n)$ as follows. For each player i, the strategy λ_i plays according to the (observation of the) path ρ in $\mathcal{G}$, as long as the previous observations follow ρ. If an observation is unexpected for Player i (i.e., differs from the sequence in ρ), then λ_i switches to an observation-based retaliating strategy λ_i^R (we will show that such a strategy exists as a consequence of (F_2)). This is a well-defined profile and a DE because: (1) all strategies are observation-based, and the outcome of the profile is the path ρ that satisfies all objectives; (2) if no deviation from the observation of ρ is detected by Player i, then by condition (F_3) we know that if the outcome does not satisfy φ_i, then it does not satisfies φ_j, for all $1 \leq j \leq n$, (3) if a deviation from the observation of ρ is detected by Player i, then the sequence of observations of Player i so far can be decomposed as $\kappa = \kappa_1(o_1, \sigma_1) \ldots (o_m, \sigma_n m$ where (o_1, σ_1) is the first deviation of the observation of ρ, and (o_m, σ_m) is the first time it is Player i's turn to play after this deviation (so possibly $m = 1$). By condition (F_2), we know that κ_1 is good for retaliation. Clearly, $\kappa_1(o_1, \sigma_1) \ldots (o_\ell, \sigma_\ell)$ is retaliation compatible as well for all $\ell \in \{1, \ldots, m\}$ since retaliation goodness is preserved by player j's actions for all j. Therefore κ is good for retaliation and by definition of retaliation goodness there exists an observation-based retaliation strategy λ_i^R for Player i which ensures that that regardless of the strategies of the opponents in coalition, if the outcome does not satisfy φ_i, then for all $j \in \{1, \ldots, n\}$, it does not satisfy φ_j either.

Second, assume that there exists a DE $(\lambda_1, \ldots, \lambda_n)$ in G, and show that (F_1), (F_2) and (F_3) hold. Let ρ be the outcome of the profile $(\lambda_1, \ldots, \lambda_n)$. Then ρ satisfies (F_1) by definition of DE. Let us show that it also satisfies (F_3). By contradiction, if $\mathsf{obs}_i(\rho)$ is not doomsday compatible for Player i, then by definition, there is a path ρ' in $\mathsf{Plays}(G)$ that is compatible with the observations and actions of player i in ρ (i.e., $\mathsf{obs}_i(\rho) = \mathsf{obs}_i(\rho')$), but ρ' does not satisfy φ_i, while it satisfies φ_j for some $j \neq i$. Then, given the strategy λ_i from the profile, the other players in coalition can choose actions to construct the path ρ' (since ρ and ρ' are observationally equivalent for player i, the observation-based strategy λ_i is going to play the same actions as in ρ). This would show that the profile is not a DE, establishing a contradiction. Hence $\mathsf{obs}_i(\rho)$ is doomsday compatible for Player i for all $i = 1, \ldots, n$ and (F_3) holds. Let us show that ρ also satisfies (F_2). Assume that this not true. Assume that κ is a prefix of $\mathsf{obs}_i(\rho)$ such that κ is not good for retaliation for Player i for some i. By definition it means that the other players can make a coalition and enforce an outcome ρ', from any prefix of play compatible with κ, that is winning for one of players of the coalition, say Player j, $j \neq i$, and losing for Player i. This contradicts the fact that λ_i belongs to a DE. □

Theorem 2. *The problem of deciding the existence of a doomsday equilibrium in an n-player game arena with imperfect information and n objectives is* EXPTIME-C *for objectives that are either all reachability, all safety, all Büchi, all co-Büchi or all parity objectives. Hardness already holds for 2-player game arenas.*

Proof. By Lemma 8, we know that we can decide the existence of a doomsday equilibrium by checking the existence of a play ρ in G that respects the conditions (F_1), (F_2), and (F_3). It can be shown (see Appendix), for all $i \in \{1, \ldots, n\}$, that the set of good for retaliation prefixes for Player i is definable by a finite-state automaton C_i, and the set of observation sequences that are doomsday compatible for Player i is definable by a Streett automaton D_i.

From the automata $(D_i)_{1 \leq i \leq n}$ and $(C_i)_{1 \leq i \leq n}$, we construct using a synchronized product a finite transition system T and check for the existence of a path in T that satisfy the winning objectives for each player in G, the Streett acceptance conditions of the $(D_i)_{1 \leq i \leq n}$, and whose all prefixes are accepted by the automata $(C_i)_{1 \leq i \leq n}$. The size of T is exponential in G and the acceptance condition is a conjunction of Streett and safety objectives. The existence of such a path can be established in polynomial time in the size of T, so in exponential time in the size of G. The EXPTIME-hardness is a consequence of the EXPTIME-hardness of two-player games of imperfect information for all the considered objectives [4,9]. $\square$

5 Conclusion

We defined the notion of doomsday threatening equilibria both for perfect and imperfect information n player games with omega-regular objectives. This notion generalizes to n player games the winning secure equilibria [10]. Applications in the analysis of security protocols are envisioned and will be pursued as future works.

We have settled the exact complexity in games of perfect information for almost all omega-regular objectives with complexities ranging from PTIME to PSPACE, the only small gap that remains is for parity objectives where we have a PSPACE algorithm and both NP and coNP-hardness. Surprisingly, the existence of doomsday threatening equilibria in n player games with imperfect information is decidable and more precisely EXPTIME-C for all the objectives that we have considered.

In a long version of this paper [8], we provide a solution in 2EXPTIME for deciding the existence of a doomsday threatening equilibrium in a game whose objectives are given as LTL formula (this solution is optimal as it is easy to show that the classical LTL realizability problem can be reduced to the DE existence problem). We also provide a Safraless solution [21] suitable to efficient implementation.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM 49, 672–713 (2002)
2. Alur, R., La Torre, S.: Deterministic generators and games for LTL fragments. TOCL 5 (2004)
3. Alur, R., La Torre, S., Madhusudan, P.: Playing games with boxes and diamonds. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 128–143. Springer, Heidelberg (2003)
4. Berwanger, D., Doyen, L.: On the power of imperfect information. In: FSTTCS, pp. 73–82 (2008)
5. Büchi, J.R., Landweber, L.H.: Definability in the monadic second-order theory of successor. J. Symb. Log. 34(2), 166–170 (1969)
6. Cai, Y., Zhang, T., Luo, H.: An improved lower bound for the complementation of rabin automata. In: LICS, pp. 167–176. IEEE Computer Society (2009)
7. Chadha, R., Kremer, S., Scedrov, A.: Formal analysis of multiparty contract signing. J. Autom. Reasoning 36(1-2), 39–83 (2006)
8. Chatterjee, K., Doyen, L., Filiot, E., Raskin, J.-F.: Doomsday equilibria for omega-regular games. CoRR, abs/1311.3238 (2013)
9. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games with imperfect information. LMCS 3(3) (2007)
10. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Games with secure equilibria. Theor. Comput. Sci. 365(1-2), 67–82 (2006)
11. Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. Inf. Comput. 208(6), 677–693 (2010)

12. Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 153–167. Springer, Heidelberg (2007)
13. Da Costa Lopes, A., Laroussinie, F., Markey, N.: ATL with strategy contexts: Expressiveness and model checking. In: FSTTCS. LIPIcs, vol. 8, pp. 120–132 (2010)
14. Emerson, E.A., Jutla, C.: Tree automata, mu-calculus and determinacy. In: FOCS, pp. 368–377. IEEE Comp. Soc. (1991)
15. Emerson, E.A., Lei, C.-L.: Modalities for model checking: Branching time strikes back. In: POPL, pp. 84–96 (1985)
16. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 190–204. Springer, Heidelberg (2010)
17. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games, vol. 2500. Springer (2002)
18. Immerman, N.: Number of quantifiers is better than number of tape cells. Journal of Computer and System Sciences 22, 384–406 (1981)
19. Jamroga, W., Mauw, S., Melissen, M.: Fairness in non-repudiation protocols. In: Meadows, C., Fernandez-Gago, C. (eds.) STM 2011. LNCS, vol. 7170, pp. 122–139. Springer, Heidelberg (2012)
20. Kremer, S., Raskin, J.-F.: A game-based verification of non-repudiation and fair exchange protocols. Journal of Computer Security 11(3), 399–430 (2003)
21. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS (2005)
22. Martin, D.: Borel determinacy. Annals of Mathematics 102, 363–371 (1975)
23. Mogavero, F., Murano, A., Perelli, G., Vardi, M.Y.: What makes ATL* decidable? A decidable fragment of strategy logic. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 193–208. Springer, Heidelberg (2012)
24. Mogavero, F., Murano, A., Vardi, M.Y.: Reasoning about strategies. In: Proc. of FSTTCS. LIPIcs, vol. 8, pp. 133–144, Schloss Dagstuhl - LZfI (2010)
25. Nash, J.F.: Equilibrium points in n-person games. PNAS 36, 48–49 (1950)
26. Piterman, N.: From nondeterministic Büchi and streett automata to deterministic parity automata. Logical Methods in Computer Science 3(3) (2007)
27. Piterman, N., Pnueli, A.: Faster solutions of rabin and streett games. In: LICS, pp. 275–284 (2006)
28. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989)
29. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. 141, 1–35 (1969)
30. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM Journal on Control and Optimization 25(1), 206–230 (1987)
31. Shapley, L.S.: Stochastic games. PNAS 39, 1095–1100 (1953)
32. Ummels, M., Wojtczak, D.: The complexity of nash equilibria in stochastic multiplayer games. Logical Methods in Computer Science 7(3) (2011)
33. Wang, F., Huang, C.-H., Yu, F.: A temporal logic for the interaction of strategies. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 466–481. Springer, Heidelberg (2011)

A Additional Details – Doomsday Equilibria for Imperfect Information Games

We present automata constructions to recognise sequences of observations that are doomsday compatible and prefixes that are good for retaliation.

Lemma 9. *Given an n-player game G with imperfect information and a set of reachability, safety or parity objectives $(\varphi_i)_{1\leq i\leq n}$, we can construct for each Player i, in exponential time, a deterministic Streett automaton D_i whose language is exactly the set of sequences of observations $\eta \in (O_i \times (\Sigma \cup \{\tau\}))^\omega$ that are* doomsday compatible *for Player i, i.e.*

$$L(D_i) = \{\eta \in (O_i \times (\Sigma \cup \{\tau\}))^\omega \mid \forall \rho \in \gamma_i(\eta) \cdot \rho \in \varphi_i \cup \bigcap_{j\neq i} \overline{\varphi_j}\}.$$

For each D_i, the size of its set of states is bounded by $\mathbf{O}(2^{nk\log k})$ and the number of Streett pairs is bounded by $\mathbf{O}(nk^2)$ where k is the number of states in G.

Proof. Let $G = (S, (S_i)_{1\leq i\leq n}, s_{\mathsf{init}}, \Sigma, \Delta, (O_i)_{1\leq i\leq n})$, and let us show the constructions for Player i, $1 \leq i \leq n$. We treat the three types of winning conditions as follows.

We start with safety objectives. Assume that the safety objectives are defined implicitly by the following tuple of sets of safe states: $(T_1, T_2, \ldots, T_n)$, i.e. $\varphi_i = \mathsf{safe}(T_i)$. First, we construct the automaton

$$A = (Q^A, q^A_{\mathsf{init}}, (O_i \times (\Sigma \cup \{\tau\}), \delta^A)$$

over the alphabet $O_i \times (\Sigma \cup \{\tau\})$ as follows:

- $Q^A = S$, i.e. the states of A are the states of the game structure G,
- $q^A_{\mathsf{init}} = s_{\mathsf{init}}$,
- $(q, (o, \sigma), q') \in \delta^A$ if $q \in o$ and there exists $\sigma' \in \Sigma$ such that $\Delta(q, \sigma') = q'$ and such that $\sigma = \tau$ if $q \notin S_i$, and $\sigma = \sigma'$ if $q \in S_i$.

The acceptance condition of A is *universal* and expressed with LTL syntax:

A word w is accepted by A iff *all* runs ρ of A on w satisfy $\rho \models \Box T_i \vee \bigwedge_{j\neq i} \Diamond \overline{T_j}$.

Clearly, the language defined by A is exactly the set of sequences of observations $\eta \in (O_i \times (\Sigma \cup \{\tau\}))^\omega$ that are *doomsday compatible* for Player i, this is because the automaton A checks (using universal nondeterminism) that all plays that are compatible with a sequence of observations are doomsday compatible.

Let us show that we can construct a deterministic Streett automaton D_i that accepts the language of A and whose size is such that: (i) its number of states is at most $\mathbf{O}(2^{(nk\log k)})$ and (ii) its number of Streett pairs is at most $\mathbf{O}(nk)$. We obtain D with the following series of constructions:

- First, note that we can equivalently see A as the intersection of the languages of $n-1$ universal automata A_j with the acceptance condition $\Box T_i \vee \Diamond \overline{T_j}$, $j \neq i$, $1 \leq j \leq n$.
- Each A_j can be modified so that a violation of T_i is made permanent and a visit to $\overline{T_j}$ is recorded. For this, we use a state space which is equal to $Q^A \times \{0,1\} \times \{0,1\}$, the first bit records a visit to $\overline{T_i}$ and the second a visit to $\overline{T_j}$. We denote this automaton by A'_j, and its acceptance condition is now $\Box\Diamond(Q^A \times \{0,1\} \times \{0\}) \rightarrow \Box\Diamond(Q^A \times \{0\} \times \{0,1\})$. Clearly, this is a universal Streett automaton with a single Streett pair.

- A'_j, which is a universal Streett automaton, can be complemented (by duality) by interpreting it as a nondeterministic Rabin automaton (with one Rabin pair). This nondeterministic Rabin automaton can be made deterministic using a Safra like procedure, and according to [6] we obtain a deterministic Rabin automaton with $\mathbf{O}(2^{k \log k})$ states and $\mathbf{O}(k)$ Rabin pairs. Let us call this automaton A''_j.
- Now, A''_j can be complemented by considering its Rabin pairs as Streett pairs (by dualization of the acceptance condition): we obtain a deterministic Streett automaton with $\mathbf{O}(k)$ Streett pairs for each A_j.
- Now, we need to take the intersection of the $n-1$ deterministic automata A''_j (interpreted as Streett automata). Using a classical synchronized product we obtain a single deterministic Streett automaton D_i of size with $\mathbf{O}(2^{nk \log k})$ states and $\mathbf{O}(nk)$ Streett pairs. This finishes our proof for safety objectives.

Let us now consider reachability objectives. Therefore we now assume the states in $T_1, \ldots, T_n$ to be target states for each player respectively, i.e. $\varphi_i = \mathsf{reach}(T_i)$. The construction is in the same spirit as the construction for safety. Let $A = (Q^A, q^A_{\mathsf{init}}, O_i \times (\Sigma \cup \{\tau\}), \delta^A)$ be the automaton over $(O_i \times (\Sigma \cup \{\tau\})$ constructed from G as for safety, with the following (universal) acceptance condition;

A word w is accepted by A iff all runs ρ of A on w satisfy $\rho \models (\bigvee_{j \neq i} \Diamond T_j) \rightarrow \Diamond T_i$.

Clearly, the language defined by A is exactly the set of sequences of observations $\eta \in ((\Sigma \cup \{\tau\}) \times O_i)^\omega$ that are *doomsday compatible* for Player i (w.r.t. the reachability objectives). Let us show that we can construct a deterministic Streett automaton D_i that accepts the language of A and whose size is such that: (i) its number of states is at most $\mathbf{O}(2^{(nk \log k)})$ and (ii) its number of Streett pairs is at most $\mathbf{O}(nk)$. We obtain D_i with the following series of constructions:

- First, the acceptance condition can be rewritten as $\bigwedge_{j \neq i} (\Diamond T_j \rightarrow \Diamond T_i)$. Then clearly if A_j is a copy of A with acceptance condition $\Diamond T_j \rightarrow \Diamond T_i$ then $L(A) = \bigcap_{j \neq i} L(A_j)$.
- For each A_j, we construct a universal Streett automaton with one Streett pair by memorizing the visits to T_i and T_j and considering the acceptance condition $\Box \Diamond T_j \rightarrow \Box \Diamond T_i$. So, we get a universal automaton with a single Streett pair.
- Then we follow exactly the last three steps (3 to 5) of the construction for safety.

Finally, let us consider parity objectives. The construction is similar to the other cases. Specifically, we can take as acceptance condition for A the universal condition $\bigwedge_{j \neq i} (\mathsf{parity}_i \vee \overline{\mathsf{parity}_j})$, and treat each condition $\mathsf{parity}_i \vee \overline{\mathsf{parity}_j}$ separately. We dualize the acceptance condition of A, into the nondeterministic condition $\overline{\mathsf{parity}_i} \wedge \mathsf{parity}_j$. This acceptance condition can be equivalently expressed as a Streett condition with at most $\mathbf{O}(k)$ Streett pairs. This automaton accepts exactly the set of observation sequences that are not doomsday compatible for Player i against Player j. Now, using optimal procedure for determinization, we can obtain a deterministic Rabin automaton, with $\mathsf{O}(k^2)$ pairs that accepts the same language [26]. Now, by interpreting the pairs of the acceptance condition as Streett pairs instead of Rabin pairs, we obtain a deterministic Streett automaton A_j that accepts the set of observations sequences that are doomsday compatible for Player i against Player j. Now, it suffices to take the product of the $n-1$

deterministic Streett automata A_j to obtain the desired automaton A, its size is at most $\mathbf{O}(2^{nk\log k})$ with at most $\mathbf{O}(nk^2)$ Streett pairs. □

Lemma 10. *Given an n-player game arena G with imperfect information and a set of reachability, safety or parity objectives $(\varphi_i)_{1\leq i\leq n}$, for each Player i, we can construct a finite-state automaton C_i that accepts exactly the prefixes of observation sequences that are good for retaliation for Player i.*

Proof. Let us show how to construct this finite-state automaton for any Player i, $1 \leq i \leq n$. Our construction follows these steps:

– First, we construct from G, according to Lemma 9, a deterministic Streett automaton $D_i = (Q^{D_i}, q^{D_i}_{\mathsf{init}}, (O_i \times (\Sigma \cup \{\tau\}), \delta^{D_i}, \mathsf{St}^{D_i})$ that accepts exactly the set of sequences of observations $\eta \in (O_i \times (\Sigma \cup \{\tau\}))^\omega$ that are *doomsday compatible* for Player i. The number of states in D_i is $\mathbf{O}(2^{|S|^2 \log|S|})$ and the number of Streett pairs is bounded by $\mathbf{O}(|S|^2 \cdot n)$, where $|S|$ is the number of states in G.
– Second, we consider a turn-based game played on D_i by two players, A and B, that move a token from states to states along edges of D_i as follows:
  1. initially, the token is in some state q
  2. then in each round: B chooses an observation $o \in O_i$ s.t. $\exists(q, (o, \sigma), q') \in \delta^{D_i}$. Then A chooses a transition $(q, (o, \sigma), q') \in \delta^{D_i}$ (which is completely determined by σ as D_i is deterministic), and the token is moved to q' where a new round starts.

  The objective of A is to enforce from state q an infinite sequence of states, so a run of D_i that starts in q, and which satisfies St^{D_i} the Streett condition of D_i. For each q, this can be decided in time polynomial in the number of states in D_i and exponential in the number of Streett pairs in St^{D_i}, see [27] for an algorithm with the best known complexity. Thus, the overall complexity is exponential in the size of the game structure G. We denote by $\mathsf{Win} \subseteq Q^{D_i}$ the set of states q from which A can win the game above.
– Note that if $(o_1, \sigma_1) \dots (o_m, \sigma_m)$ is the trace of a path from q_{init} in D_i to a state $q \in \mathsf{Win}$, then clearly $(o_1, \sigma_1) \dots (o_{n-1}, \sigma_{n-1})o_n$ is good for retaliation. Indeed, the winning strategy of A in q is an observation based retaliating strategy λ^R_i for Player i in G. On the other hand, if a prefix of observations reaches $q \notin \mathsf{Win}$ then by determinacy of Streett games, we know that B has a winning strategy in q and this winning strategy is a strategy for the coalition (against Player i) in G to enforce a play in which Player i does not win and at least one of the other players wins. So, from D_i and Win, we can construct a finite state automaton C_i which is obtained as a copy of D_i with the following acceptance condition: a prefix $\kappa = (o_0, \sigma_0), (o_1, \sigma_1), \dots, (o_{k-1}, \sigma_{k-1}), o_k$ is accepted by C_i if there exists a path $q_0 q_1 \dots q_k$ in C_i such that q_0 is the initial state of C_i and either there exists a transition labeled (o_k, σ) from q_k to a state of Win. □

Bisimulations and Logical Characterizations on Continuous-Time Markov Decision Processes

Lei Song[1], Lijun Zhang[2], and Jens Chr. Godskesen[3]

[1] Max-Planck-Institut für Informatik and Saarland University, Saarbrücken, Germany
[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[3] Programming, Logic, and Semantics Group, IT University of Copenhagen, Denmark

Abstract. In this paper we study strong and weak bisimulation equivalences for continuous-time Markov decision processes (CTMDPs) and the logical characterizations of these relations with respect to the continuous-time stochastic logic (CSL). For strong bisimulation, it is well known that it is strictly finer than the CSL equivalence. In this paper we propose strong and weak bisimulations for CTMDPs and show that for a subclass of CTMDPs, strong and weak bisimulations are both sound and complete with respect to the equivalences induced by CSL and the sub-logic of CSL without next operator respectively. We then consider a standard extension of CSL, and show that it and its sub-logic without X can be fully characterized by strong and weak bisimulations respectively over arbitrary CTMDPs.

1 Introduction

Recently, continuous-time Markov decision processes (CTMDPs) have received extensive attention in the model checking community, see for example [5,37,26,27,12,30]. Analysis techniques for CTMDPs suffer especially from the state space explosion problem. Thus, as for other stochastic models, bisimulation relations have been proposed for CTMDPs. In [26], strong bisimulation was shown to be sound with respect to the continuous-time stochastic logic [2] (CSL). This result guarantees that one can first reduce a CTMDP up to bisimulation equivalence before analysing it. On the other hand, as indicated in [26], strong bisimulation is not complete with respect to CSL, i.e., logically equivalent states might be not bisimilar.

CTMDPs extend Markov decision processes (MDPs) with exponential sojourn time distributions, and subsume models such as labelled transition systems and Markov chains as well. While linear and branching time equivalences have been studied for these sub-models [36,35,6,33], we extend these results to the setting of CTMDPs. In this paper we study strong and weak bisimulation relations for CTMDPs, and the logical characterization problem of these relations with respect to CSL and its sub-logics.

We start with a slightly coarser notion of strong bisimulation than the one in [26], and then propose *weak bisimulation* for CTMDPs. We study the relationship between strong and weak bisimulations and the logical equivalences induced by CSL and $\mathrm{CSL}_{\setminus\mathsf{X}}$ – the sub-logic of CSL without next operators. Our first contribution is to identify a subclass of CTMDPs under which our strong and weak bisimulations coincide with CSL and

$\mathrm{CSL}_{\backslash \mathrm{X}}$ equivalences respectively. We discuss then how this class of CTMDPs can be efficiently determined, and moreover, we argue that most models arising in practice are among this class.

As for labelled transition systems and MDPs, we also define an extension of CSL, called CSL^*, which is more distinguishable than CSL. Surprisingly, CSL^* is able to fully characterize strong bisimulation over arbitrary CTMDPs, similarly for the sub-logic without next operator and weak bisimulation.

Since CTMDPs can be seen as models combining both MDPs and continuous-time Markov chains (CTMCs), we will discuss the downward compatibility of the relations with those for MDPs [31] and CTMCs in [6]. Summarizing, the paper contains the following contributions:

1. We extend strong probabilistic bisimulation defined in [31] over probabilistic automata to CTMDPs, and then prove that it coincides with CSL equivalence for a subclass of CTMDPs;
2. We propose a scheme to determine the subclass of CTMDPs efficiently, and show that many models in practice are in this subclass;
3. We introduce a new notion of weak bisimulation for CTMDPs, and show its characterization results with respect to $\mathrm{CSL}_{\backslash \mathrm{X}}$;
4. We present a standard extension of CSL that is shown to be both sound and complete with respect to strong and weak bisimulations for arbitrary CTMDPs.

Related work. Logical characterizations of bisimulations have been studied extensively for stochastic models. For CTMCs, CSL characterizes strong bisimulation, while CSL without next operator characterizes weak bisimulation [6]. Our results in this paper are conservative extensions for both strong and weak bisimulations from CTMCs to CTMDPs. In [17], the results are extended to CTMCs with continuous state spaces.

For CTMDPs, the first logical characterization result is presented in [26]. It is shown that strong bisimulation is sound, but not complete with respect to CSL equivalence. In this paper, we introduce strong and weak bisimulation relations for CTMDPs. For a subclass of CTMDPs, i.e., those without *2-step recurrent* states, we show that they are also complete for CSL and $\mathrm{CSL}_{\backslash \mathrm{X}}$ equivalences respectively.

For probabilistic automata (PAs), Hennessy-Milner logic has been extended to characterize bisimulations in [22,14,20]. In [16], Desharnais *et al.* have shown that weak bisimulation agrees with PCTL^* equivalence for alternative PAs. Another related paper for PAs is our previous paper [33], in which we have introduced i-depth bisimulations to characterize logical equivalences induced by PCTL^* and its sub-logics.

All proofs are found in the full version of this paper [34].

Organization of the paper. Section 2 recalls the definition of CTMDPs and the logic CSL. Variants of bisimulation relations and their corresponding logical characterization results are studied in Section 3. In Section 4 we present the extension of CSL that fully characterizes strong and weak bisimulations. We discuss in Section 5 related work with MDPs and CTMCs. Section 6 concludes the paper.

2 Preliminaries

For a finite set S, a distribution is a function $\mu : S \to [0,1]$ satisfying $|\mu| := \sum_{s \in S} \mu(s) = 1$. We denote by $Dist(S)$ the set of distributions over S. We shall use $s, r, t, \ldots$ and $\mu, \nu \ldots$ to range over S and $Dist(S)$, respectively. The support of μ is defined by $Supp(\mu) = \{s \in S \mid \mu(s) > 0\}$. Given a finite set of non-negative real numbers $\{p_j\}_{j \in J}$ and distributions $\{\mu_j\}_{j \in J}$ such that $\sum_{j \in J} p_i = 1$ for each $j \in J$, $\sum_{j \in J} p_j \cdot \mu_j$ is the distribution such that $(\sum_{j \in J} p_j \cdot \mu_j)(s) = \sum_{j \in J} p_j \cdot \mu_j(s)$ for each $s \in S$. For an equivalence relation $\mathcal{R}$ over S, we write $\mu \mathrel{\mathcal{R}} \nu$ if it holds that $\mu(C) = \nu(C)$ for all equivalence classes $C \in S/\mathcal{R}$ where $\mu(C) = \sum_{s \in C} \mu(s)$, and moreover $[s]_{\mathcal{R}} = \{r \mid s \mathrel{\mathcal{R}} r\}$ is the equivalence class of $S/\mathcal{R}$ containing s. The subscript $\mathcal{R}$ will be omitted if it is clear from the context. A distribution μ is called *Dirac* if $|Supp(\mu)| = 1$, and we let $\mathcal{D}_s$ denote the Dirac distribution such that $\mathcal{D}_s(s) = 1$. We let $R^{\geq 0}$ and $R^{>0}$ denote the set of non-negative and positive real numbers respectively.

2.1 Continuous-Time Markov Decision Processes

Below follows the definition of CTMDPs, which subsume both MDPs and CTMCs.

Definition 1 (Continuous-Time Markov Decision Processes). *A tuple $\mathcal{C} = (S, \to, AP, L, s_0)$ is a* CTMDP *where $s_0 \in S$ is the initial state, S is a finite but non-empty set of states, AP is a finite set of atomic propositions, $L : S \mapsto 2^{AP}$ is a labelling function, and $\to \subseteq S \times R^{>0} \times Dist(S)$ is a finite transition relation such that for each $s \in S$, there exists λ and μ with $(s, \lambda, \mu) \in \to$.*

From Definition 1 we can see that there are both non-deterministic and probabilistic transitions in a CTMDP. We write $s \xrightarrow{\lambda} \mu$ if $(s, \lambda, \mu) \in \to$, where λ is called exit rate of the transition. Let $Suc(s) = \{r \mid \exists (s \xrightarrow{\lambda} \mu).\mu(r) > 0\}$ denote the successor states of s, and let $Suc^*(s)$ be its transitive closure. A state s is said to be *silent* iff for all $s_1, s_2 \in Suc^*(s)$, $L(s_1) = L(s_2)$ and $s_1 \xrightarrow{\lambda} \mu_1$ implies $s_2 \xrightarrow{\lambda} \mu_2$. Intuitively, a state s is silent if all its reachable states have the same labels as s. In addition, they have transitions with the same exit rates as transitions of s. States like s are called silent, since it is not distinguishable from all its successors, either by labels or sojourn time of states. Therefore a silent state s and all its successors can be represented by a single state which is the same as s but with all its outgoing transitions leading to itself. A CTMC is a deterministic CTMDP satisfying the condition: $s \xrightarrow{\lambda} \mu$ and $s \xrightarrow{\lambda'} \mu'$ imply $\lambda = \lambda'$ and $\mu = \mu'$ for any $s \in S$.

2.2 Paths, Uniformization, and Measurable Schedulers

Let $\mathcal{C} = (S, \to, AP, L, s_0)$ be a CTMDP fixed for the remainder of the paper. Let $Paths^n(\mathcal{C}) = S \times (R^{>0} \times S)^n$ denote the set containing paths of $\mathcal{C}$ with length n. The set of all finite paths of $\mathcal{C}$ is the union of all finite paths $Paths^*(\mathcal{C}) = \cup_{n \geq 0} Paths^n(\mathcal{C})$. Moreover, $Paths^\infty(\mathcal{C}) = S \times (R^{>0} \times S)^\infty$ contains all infinite paths and $Paths(\mathcal{C}) = Paths^*(\mathcal{C}) \cup Paths^\infty(\mathcal{C})$ is the set of all (finite and infinite) paths of $\mathcal{C}$. Intuitively, a path is comprised of an alternation of states and their sojourn time. To simplify the discussion we introduce some notations. Given a path $\omega = s_0, t_0, s_1, t_1 \cdots s_n \in Paths^n(\mathcal{C})$, $|\omega| = n$ is the length of ω, $\omega \downarrow = s_n$ is the last state of ω, $\omega|^i = s_0, t_0, \cdots, s_i$ is the prefix of ω ending at the $(i+1)$-th state, and $\omega|_i = s_i, t_i, s_{i+1}, \cdots$ is the suffix of ω starting from the $(i+1)$-th state, and $\omega \frown (t_n, s_{n+1})$ is the path obtained by extending ω with (t_n, s_{n+1}). Let $\omega[i] = s_i$ denote the $(i+1)$-th state where $i \leq n$ and $time(\omega, i) = t_i$ the sojourn time in the $(i+1)$-th state with $i < n$. Let $\omega@t$ be the state at time t in ω, that is, $\omega@t = \omega[j]$ where j is the smallest index such that $\sum_{i=0}^{j} t_i > t$. Moreover, $Steps(s) = \{(\lambda, \mu) \mid (s, \lambda, \mu) \in \rightarrow\}$ is the set of all available choices at state s. Let $\{I_i \subseteq [0, \infty)\}_{0 \leq i \leq k}$ denote a set of non-empty closed intervals, then $C(s_0, I_0, \cdots, I_k, s_{k+1})$ is the *cylinder set* of paths $\omega \in Paths^{\infty}(\mathcal{C})$ such that $\omega[i] = s_i$ for $0 \leq i \leq k+1$ and $time(\omega, i) \in I_i$ for $0 \leq i \leq k$. Let $\mathfrak{F}_{Paths^{\infty}(\mathcal{C})}$ be the smallest σ algebra on $Paths^{\infty}(\mathcal{C})$ containing all cylinder sets.

As shown in [4], model checking of CTMCs can be reduced to the problem of computing transient state probabilities, which can be solved efficiently, for instance by uniformization. In a uniformized CTMC, all states will evolve at the same speed, i.e., all transitions have the same exit rates. Similarly, we can also define uniformization of a CTMDP by uniformizing the exit rate of all its transitions. Below we recall the notion of *uniformization* for CTMDPs [12,27].

Definition 2 (Uniformization). *Given a* CTMDP $\mathcal{C} = (S, \rightarrow, AP, L, s_0)$*, the uniformized* CTMDP *is denoted as* $\bar{\mathcal{C}} = (\bar{S}, \rightarrow', AP, \bar{L}, \bar{s_0})$ *where*

1. $\bar{S} = \{\bar{s} \mid s \in S\}$, $\bar{s_0} \in \bar{S}$ *is the initial state,*
2. $\bar{L}(\bar{s}) = L(s)$ *for each* $s \in S$*, and*
3. $(\bar{s}, E, \bar{\mu}) \in \rightarrow'$ *iff there exists* $(s, \lambda, \mu) \in \rightarrow$ *and* $\bar{\mu} = \frac{\lambda}{E} \cdot \mu' + (1 - \frac{\lambda}{E}) \cdot \mathcal{D}_{\bar{s}}$ *such that* $\mu'(\bar{r}) = \mu(r)$ *for each* $r \in Supp(\mu)$*,*

Here E is the uniformization rate for $\bar{\mathcal{C}}$, which is a real number equal or greater than all the rates appearing in $\mathcal{C}$.

By uniformization for each transition (s, λ, μ) we add a self loop to s with rate equal to E minus the original rate λ. After uniformization every state will have a unique exit rate on all its transitions. As we will show later, this transformation will not change the properties we are interested in under certain classes of schedulers.

Due to the existence of non-deterministic choices in CTMDPs, we need to resolve them to define probability measures. As usual, non-deterministic choices in CTMDPs are resolved by schedulers (or policies or adversaries), which generate a distribution over the available transitions based on the given history information. Different classes of schedulers can be defined depending on the information a scheduler can use in order to choose the next transition. However not all of them are suitable for our purposes, which we will explain later. In this paper, we shall focus on one specific class of schedulers, called *measurable total time positional schedulers* (TTP) [27], which is defined as follows:

Definition 3 (Schedulers). *A scheduler* $\pi : S \times R^{\geq 0} \times (R^{>0} \times Dist(S)) \mapsto [0, 1]$ *is measurable if* $\pi(s, t, \cdot) \in Dist(Steps(s))$ *for all* $(s, t) \in S \times R^{\geq 0}$ *and* $\pi(\cdot, tr)$ *are measurable for all* $tr \in 2^{(R^{>0} \times Dist(S))}$*, where*

– $\pi(s,t,\cdot)$ *is a distribution such that* $\pi(s,t,\cdot)(\lambda,\mu) = \pi(s,t,\lambda,\mu)$*, and*
– $\pi(\cdot,tr) : (S \times R^{\geq 0}) \mapsto [0,1]$ *is a function such that for each* $(s,t) \in S \times R^{\geq 0}$*, it holds* $\pi(\cdot,tr)(s,t) = \sum_{(\lambda,\mu)\in tr} \pi(s,t,\lambda,\mu)$.

The schedulers defined in Definition 3 are total time positional, since they make decisions only based on the current state and total elapsed time, which are the first and second parameters of π respectively. The third parameter and fourth parameter of π denote the rate and the resulting distribution of the chosen transition respectively. Given the current state s, the total elapsed time t, and a transition (λ,μ), π will return the probability with which (λ,μ) will be chosen. This is a special case of the general definition of schedulers, which can make decisions based on the full history, for instance visited states and the sojourn time at each state. Given a scheduler π, a unique probability measure $Pr_{\pi,s}$ can be determined on the σ-algebra $\mathfrak{F}_{Paths^\infty(\mathcal{C})}$ inductively as below: $Pr_{\pi,s}(C(s_0, I_0, \cdots, s_n), tt) =$

$$\begin{cases} 1 & n = 0 \wedge s = s_0 \quad \text{(1a)} \\ 0 & s \neq s_0 \quad \text{(1b)} \\ \int\limits_{t\in I_0} \sum\limits_{(\lambda,\mu)\in tr} \pi(s_0,tt)(\lambda,\mu)\cdot\mu(s_1)\cdot\lambda e^{-\lambda t}\cdot Pr_{\pi,s_1}dt & \text{otherwise} \quad \text{(1c)} \end{cases}$$

where Pr_{π,s_1} is an abbreviation of $Pr_{\pi,s_1}(C(s_1,\ldots,s_n), tt+t)$, $tr = Steps(s_0)$ and tt is the parameter denoting the total elapsed time. One nice property of TTP schedulers is that uniformization does not change time-bounded reachabilities under TTP schedulers [27,30]. This result can be extended to cover more properties like CSL$_{\backslash\mathsf{X}}$ and CSL$^*_{\backslash\mathsf{X}}$, which shall be introduced soon.

Besides TTP schedulers, there are other different classes of schedulers for CTMDPs, some of which are insensitive to uniformization, whereas some of which may gain or lose information after uniformization, i.e., properties of a CTMDP may be changed by uniformization. To avoid technical overhead in the presentation, we refer to [27] for an in-depth discussion of these different classes of schedulers and their relation to uniformization.

2.3 Continuous Stochastic Logic

Logical formulas are important for verification purpose, since they offer a rigorous and unambiguous way to express properties one may want to check. Probabilistic computation tree logic (PCTL) [18] is often used to express properties of probabilistic systems. In order to deal with probabilistic systems with exponential sojourn time distributions like CTMCs and CTMDPs, the continuous stochastic logic (CSL) was introduced to reason about CTMCs [2,4], and recently extended to reason about CTMDPs in [26]. CSL contains both state[1] and path formulas whose syntax is defined by the following BNFs:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{P}_{\bowtie p}(\psi),$$
$$\psi ::= \mathsf{X}^I\,\varphi \mid \varphi\mathsf{U}^I\varphi,$$

[1] The steady-state operator is omitted in this paper for simplicity of presentation.

where $a \in AP$, $p \in [0,1]$, $\bowtie \in \{<, \leq, \geq, >\}$, and $I \subseteq [0, \infty)$ is a non-empty closed interval.

We use $s \models \varphi$ to denote that s satisfies the state formula φ, while $\omega \models \psi$ denotes that ω satisfies the path formula ψ. The satisfaction relation for atomic proposition and Boolean operators is standard. Below we give the satisfaction relation for the remaining state and path formulas:

$$s_0 \models \mathcal{P}_{\bowtie p}(\psi) \text{ iff } \forall \pi . Pr_{\pi, s_0}(\{\omega \in Paths^{\infty}(\mathcal{C}) \mid \omega \models \psi\}) \bowtie p,$$

$$\omega \models \mathsf{X}^I \varphi \text{ iff } \omega[1] \models \varphi \wedge time(\omega, 0) \in I,$$

$$\omega \models \varphi_1 \mathsf{U}^I \varphi_2 \text{ iff } \exists i . (\sum_{0 \leq j < i} time(\omega, j) \in I \wedge \omega[i] \models \varphi_2 \wedge (\forall 0 \leq j < i . \omega[j] \models \varphi_1)).$$

Intuitively, a state s_0 satisfies $\mathcal{P}_{\bowtie p}(\psi)$ iff no matter how we schedule the transitions of s_0 and its successors, the probability of paths starting from s_0 and satisfying ψ is always $\bowtie p$. This operator has the same semantics as in PCTL. Compared to PCTL, the main difference arises in the semantics of the path formulas. Given a path ω, we say $\omega \models \mathsf{X}^I \varphi$, iff the second state in ω satisfies φ, moreover the sojourn time in the first state of ω is within the time interval I. We say $\omega \models \varphi_1 \mathsf{U}^I \varphi_2$, iff along ω, a state satisfying φ_2 can be reached at some time point in I, and all the preceding states if any satisfy φ_1. If all time bounds are defined to be equal to $[0, \infty)$, i.e., removing time restrictions, CSL will degenerate to PCTL.

Different from [4] where the semantics of CSL is continuous, in this paper we consider pointwise semantics of CSL. This is mainly because the semantics of CSL* introduced in Section 4 is also pointwise. However, results in Section 3 are also valid if we consider continuous semantics.

Logic Equivalences. Let $\mathcal{L}$ denote some logic. We say that s and r are $\mathcal{L}$-equivalent, denoted by $s \sim_{\mathcal{L}} r$, if they satisfy the same set of $\mathcal{L}$ state formulas, that is, $s \models \varphi$ iff $r \models \varphi$ for all state formulas φ in $\mathcal{L}$, similarly for $\sim_{\mathcal{L}_{\setminus \mathsf{X}}}$, where $\mathcal{L}_{\setminus \mathsf{X}}$ denotes the sublogic of $\mathcal{L}$ without the X^I operator. In this paper, $\mathcal{L}$ will denote either CSL or CSL*, which we shall introduce in Section 4.

3 Bisimilarity and CSL Equivalence

In this section, we first introduce the concept of strong bisimulation for CTMDPs, which can be seen as a variant of strong bisimulation for MDPs. Then we define a sub-class of CTMDPs, called non 2-step recurrent CTMDPs, and show that strong bisimulation can be fully characterized by CSL for non 2-step recurrent CTMDPs. We extend the work to the weak setting and show similar results for weak bisimulation. Finally, we propose an efficient scheme to determine non 2-step recurrent CTMDPs and we show that almost all CTMDP models in practice fall into this class.

3.1 Strong Bisimulation

The definition of strong bisimulation we shall introduce in this section slightly generalizes the one introduced in [26]. The reason is that we adopt the notion of combined transitions, used in [31] to define *strong probabilistic bisimulation* for PAs. Combined transitions allow transitions induced by convex combinations of several transitions. We shall lift its definition to the setting of CTMDPs. Let $s \xrightarrow{\lambda}_{\mathrm{P}} \mu$ iff there exists $\{s \xrightarrow{\lambda} \mu_j\}_{j\in J}$ and $\{p_j\}_{j\in J}$ such that $\sum_{j\in J} p_j = 1$, and $\sum_{j\in J} p_j \cdot \mu_j = \mu$. The combined transitions of a CTMDP are almost the same as those for PAs except we need to take care of the rate of each transition. Here we only allow to combine transitions with the same rate, otherwise we may change non-trivial properties of a CTMDP, which we will explain soon. Below follows the definition of strong bisimulation:

Definition 4 (Strong Bisimulation). *Let $\mathcal{R} \subseteq S \times S$ be an equivalence relation. $\mathcal{R}$ is a strong bisimulation iff $s\ \mathcal{R}\ r$ implies that $L(s) = L(r)$ and for each $s \xrightarrow{\lambda} \mu$, there exists $r \xrightarrow{\lambda}_{P} \mu'$ such that $\mu\ \mathcal{R}\ \mu'$.*

We write $s \sim r$ whenever there exists a strong bisimulation $\mathcal{R}$ such that $s\ \mathcal{R}\ r$. Let strong bisimilarity $\sim$ denote the largest strong bisimulation, which is equal to the union of all strong bisimulation relations.

For s and r to be strong bisimilar, the same set of atomic propositions should hold at s and r. Furthermore, s should be able to mimic r stepwise and vice versa, that is, whenever s has a transition with label λ leading to a distribution μ, r should also be able to perform a (combined) transition with the same label to a distribution ν such that μ and ν match with each other, i.e., μ and ν assign the same probability to each equivalence class $C \in S/\mathcal{R}$. Strong bisimulation defined in Definition 4 is a conservative extension of strong probabilistic bisimulation for PAs defined in [31], in the sense that it coincides with strong probabilistic bisimulation if we replace λ with actions.

The relation defined above is slightly coarser than the one considered in [26], where the combined transition $r \xrightarrow{\lambda}_{\mathrm{P}} \mu'$ is replaced by the normal transition $r \xrightarrow{\lambda} \mu'$. In [26], it was also shown that strong bisimulation is only sound but not complete with respect to CSL equivalence. Even though our definition of strong bisimulation is slightly coarser, it is still too fine for CSL equivalence as shown in the following theorem:

Theorem 1 ([26]). $\sim\ \subsetneq\ \sim_{CSL}$.

The proof in [26] can be directly adapted to prove the soundness of our slightly more general strong bisimulation. The inclusion in Theorem 1 is strict which is illustrated by the following example:

Example 1. Suppose we are given two states s_0 and r_0 of a CTMDP depicted in Fig. 1 (a) and (b) respectively, where all states have different atomic propositions except $L(s_0) = L(r_0)$. Assume u_i are silent for $i = 1, 2, 3$, our aim is to show that s_0 and r_0 satisfy the same set of CSL formulas, while they are not strong bisimilar by Definition 4.

We first show that $s_0 \sim_{\mathrm{CSL}} r_0$, i.e., $s_0 \models \varphi$ implies $r_0 \models \varphi$ for any φ and vice versa. The only non-trivial cases are the time-bounded reachabilities from s_0 and r_0 to states in $C \subseteq \{u_1, u_2, u_3\}$. For instance the maximal probability from s_0 and r_0 to $\{u_2, u_3\}$ in time interval $[a, b]$ is equal to $0.7 \cdot (e^{-a} - e^{-b})$, irrelevant of the middle transition of r_0. Similarly, we can check that for other C, the maximal (or minimal) probabilities from s_0 and r_0 to C in time interval I are all independent from the middle transition of r_0. Therefore we conclude that $s_0 \sim_{\mathrm{CSL}} r_0$.

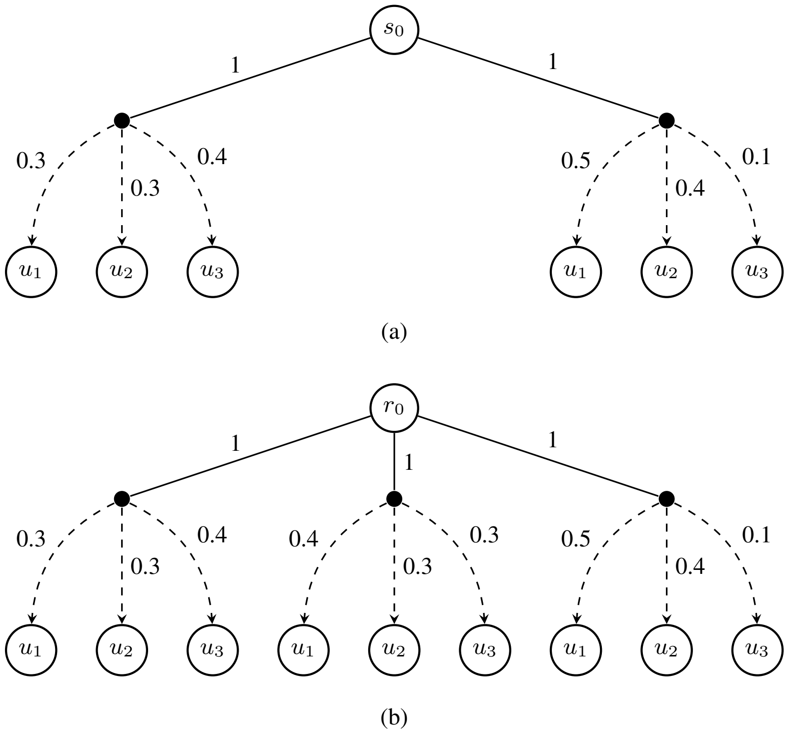


Fig. 1. Counterexample of the completeness of strong bisimulation

Secondly, we show that it does not hold that $s_0 \sim r_0$ according to Definition 4. We prove by contradiction. Assume that there exists a strong bisimulation $\mathcal{R}$ such that $s_0 \mathcal{R} r_0$. By Definition 4, for the middle transition of r_0, i.e., $r_0 \xrightarrow{1} \mu'$ where $\mu'(u_1) = 0.4, \mu'(u_2) = 0.3$, and $\mu'(u_3) = 0.3$, we need to find a transition $s_0 \xrightarrow{1}_{\text{P}} \mu$ of s_0 such that $\mu \mathcal{R} \mu'$. Since u_1, u_2, and u_3 have different atomic propositions, $(u_i, u_j) \notin \mathcal{R}$ for any $1 \leq i \neq j \leq 3$. Therefore the only possibility is that $\mu(u_1) = 0.4, \mu(u_2) = 0.3$, and $\mu(u_3) = 0.3$. However that is impossible, such μ cannot be the resulting distribution of any (combined) transition of s_0. Otherwise there would exist $w_1, w_2 > 0$ such that $w_1 + w_2 = 1$, $0.3 \cdot w_1 + 0.5 \cdot w_2 = 0.4$, and $0.3 \cdot w_1 + 0.4 \cdot w_2 = 0.3$ according to the definition of combined transition, which is clearly not possible. Hence we conclude that $s_0 \not\sim r_0$, and $\sim$ is finer than $\sim_{\text{CSL}}$. □

In [30] randomized schedulers allow to combine transitions with different rates, i.e., the combined transition is defined as: $s \xrightarrow{\lambda}_{\text{P}} \mu$ iff there exist $\{s \xrightarrow{\lambda_i} \mu_i\}_{i \in I}$ and $\{p_i\}_{i \in I}$ such that $\sum_{i \in I} p_i \cdot \lambda_i = \lambda$ and $\sum_{i \in I} p_i \cdot \mu_i = \mu$, where $p_i \in [0, 1]$ for each $i \in I$ and $\sum_{i \in I} p_i = 1$. By adopting this definition of combined transition in Definition 4, we will obtain a coarser strong bisimulation. However it turns out that this new definition of strong bisimulation is too coarse for CSL equivalence, since there exist two states which are strong bisimilar according to the new definition, but they satisfy different CSL formulas. Refer to the following example:

Example 2. Suppose that we have two states s_1 and r_1 such that s_1 has two non-deterministic transitions which can evolve into u_1 with rates 1 or 4 respectively. The state r_1 is the same as s_1 except that it can evolve into u_1 with an extra transition of rate 2. Also we assume that $L(s_1) = L(r_1)$ and u_1 is a silent state with $L(u_1) \not\subseteq L(s_1)$. Suppose that we adopt the new definition of combined transition in Definition 4 by allowing to combine transitions with different rates, we shall show that s_1 and r_1 are strong bisimilar, but they are not CSL-equivalent.

We first show that s_1 and r_1 are strong bisimilar. Let $\mathcal{R}$ be an equivalence relation only equating s_1 and r_1, it suffices to prove that $\mathcal{R}$ is a strong bisimulation. The only non-trivial case is when $r_1 \xrightarrow{2} \mathcal{D}_{u_1}$, we need to find a matching transition of s_1. Since we allow to combine transitions of different rates, a combined transition $s_1 \xrightarrow{2}_{\mathrm{P}} \mathcal{D}_{u_1}$ can be obtained by assigning weights $\frac{2}{3}$ and $\frac{1}{3}$ to transitions $s_1 \xrightarrow{1} \mathcal{D}_{u_1}$ and $s_1 \xrightarrow{4} \mathcal{D}_{u_1}$ respectively. Therefore we conclude that s_1 and r_1 are strong bisimilar.

Secondly, we show that s_1 and r_1 are not CSL equivalent. It suffices to find a formula φ such that $s_1 \models \varphi$ but $r_1 \not\models \varphi$. Let $\psi = \mathsf{X}^{[a,b]}\, L(u_1)$ where $0 \leq a < b$. The probabilities for paths starting from s_1 and satisfying ψ by choosing the transitions with rates 1, 2, and 4 are equal to $e^{-a} - e^{-b}$, $e^{-2a} - e^{-2b}$, and $e^{-4a} - e^{-4b}$ respectively. We need only to find a and b such that $e^{-2a} - e^{-2b} > \max\{e^{-a} - e^{-b}, e^{-4a} - e^{-4b}\}$. Let $a = 0.2$ and $b = 1$, then $e^{-a} - e^{-b} \approx 0.45$, $e^{-2a} - e^{-2b} \approx 0.53$, and $e^{-4a} - e^{-4b} \approx 0.43$. Let $\varphi = \mathcal{P}_{\leq 0.46}(\mathsf{X}^{[0.2,1]}\, L(u_1))$, obviously $s_1 \models \varphi$, but $r_1 \not\models \varphi$, which means that s_1 and r_1 are not CSL-equivalent. □

Example 2 also shows that in order for two states satisfying the same CSL formulas, it is necessary for them to have transitions with the same exit rates, otherwise we can always find CSL formulas distinguishing them, which also justifies that we only allow to combine transitions with the same rate in Definition 4.

We have shown in Example 1 that $\sim$ is not complete with respect to $\sim_{\mathrm{CSL}}$. However in the sequel we shall identify a special class of CTMDPs, in which the completeness holds. We first give two examples for inspiration:

Example 3. In this example, we show that, it is impossible to construct similar states as s_0 and r_0 in Example 1 such that they are not strong bisimilar but only have 2 distinct successors.

Let s_2 and r_2 denote the two states depicted in Fig. 2, where $x \in [0, 1]$ denotes an arbitrary or unknown probability and all states have different atomic propositions except that $L(s_2) = L(r_2)$. Our aim is to show that states in form of s_2 and r_2 must be strong bisimilar, provided that $s_2 \sim_{\mathrm{CSL}} r_2$. First we show that $x \in [\frac{1}{4}, \frac{1}{2}]$ in order that $s_2 \sim_{\mathrm{CSL}} r_2$. This is done by contradiction. Assume that $x > \frac{1}{2}$ and let $\psi = \mathsf{X}^{[0,\infty)}(L(u_1))$. Then the maximal probability of paths starting from s_2 and satisfying ψ is equal to $\frac{1}{2}$, while the maximal probability of paths starting from r_2 and satisfying ψ is equal to x. Since $x > \frac{1}{2}$, $s_2 \models \mathcal{P}_{\leq\frac{1}{2}}(\psi)$, while $r_2 \not\models \mathcal{P}_{\leq\frac{1}{2}}(\psi)$, therefore $s_2 \not\sim_{\mathrm{CSL}} r_2$. Similarly, we can show that it is not possible for $x < \frac{1}{4}$, hence it holds that $x \in [\frac{1}{4}, \frac{1}{2}]$.

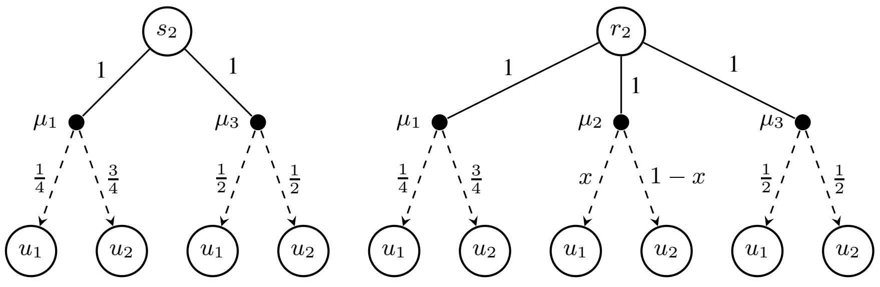


Fig. 2. s_2 can always simulate the middle transition of r_2, as long as $\frac{1}{4} \leq x \leq \frac{1}{2}$

Secondly, we show that $s_2 \sim r_2$ given that $x \in [\frac{1}{4}, \frac{1}{2}]$. Let $\mathcal{R}$ be an equivalence relation only equating s_2 and r_2, it suffices to show that $\mathcal{R}$ is a strong bisimulation according to Definition 4. Let μ_1, μ_2, and μ_3 be distributions defined in Fig. 2. The only non-trivial case is when $r_2 \xrightarrow{1} \mu_2$, we need to show that there exists w_1 and w_2 such that $w_1 + w_2 = 1$, $(w_1 \cdot \mu_1 + w_2 \cdot \mu_3)\ \mathcal{R}\ \mu_2$. Let $w_1 = 2 - 4x$ and $w_2 = 4x - 1$, it is easy to verify that $w_1, w_2 \in [0, 1]$ and $w_1 + w_2 = 1$, since $x \in [\frac{1}{4}, \frac{1}{2}]$. Moreover, $w_1 \cdot \mu_1 + w_2 \cdot \mu_3 = \mu_2$, since $w_1 \cdot \frac{1}{4} + w_2 \cdot \frac{1}{2} = x$ and $w_1 \cdot \frac{3}{4} + w_2 \cdot \frac{1}{2} = 1 - x$. Therefore $s_2 \xrightarrow{1}_{\mathrm{P}} \mu_2$ as desired, and $\mathcal{R}$ is indeed a strong bisimulation. □

In order for Example 1 being a valid counterexample for $\sim_{\mathrm{CSL}}\ \subseteq\ \sim$, we have made another assumption that u_i $(i = 1, 2, 3)$ are silent, i.e., they cannot evolve into other states not equivalent to themselves with positive probability. This assumption is also crucial which can be seen by the following example:

Example 4. Consider again the two states s_0 and r_0 introduced in Example 1, where we prove that s_0 and r_0 are CSL equivalent. Now suppose that u_3 is not silent, but can evolve into some state u_3' with rate 1, where u_3' is a state with different atomic propositions from all the others. We are going to show that s_0 and r_0 are not CSL equivalent anymore with this slight change. Consider the path formula: $\psi = (L(s_0) \vee L(u_3))\mathsf{U}^{[0,b]}(L(u_2) \vee L(u_3'))$, we can show that the probabilities of paths starting from r_0 and satisfying ψ by choosing the left, middle, and right transitions are equal to: $L = 0.3 \cdot w_1 + 0.4 \cdot w_2$, $M = 0.3 \cdot w_1 + 0.3 \cdot w_2$, and $R = 0.4 \cdot w_1 + 0.1 \cdot w_2$ respectively, where $w_1 = 1 - e^{-b}$ and $w_2 = 1 - e^{-b} - b \cdot e^{-b}$. It suffices to find a b such that $M < \min\{L, R\}$, which means that the middle transition of r_0 dominates the minimal probability of satisfying ψ. Such b exists, for instance, by letting $b = 1$ we obtain: $L \approx 0.295$, $M \approx 0.269$, and $R \approx 0.279$, apparently, $M < \min\{L, R\}$. In other words, let $b = 1$ in ψ, we have $s_0 \models \mathcal{P}_{\geq R}(\psi)$, but $r_0 \not\models \mathcal{P}_{\geq R}(\psi)$, since there exists a scheduler of r_0, i.e., the one choosing the middle transition of r_0 such that the probability of satisfying ψ is equal to M, which is strictly less than R. Therefore $s_0 \not\sim_{\mathrm{CSL}} r_0$. □

In Example 1, we have shown that s_0 and r_0 satisfy the same CSL formulas, but they are not strong bisimilar. However in Examples 3 and 4, we show that without the two assumptions:

- s_0 and r_0 should have more than 2 states among their successors;
- there exists no successor which can evolve into a state not CSL equivalent to other states with positive probability,

we can guarantee that either s_0 and r_0 are strong bisimilar, or they are not CSL equivalent. These intuitions lead us to the special class of CTMDPs, which we call *non 2-step recurrent* CTMDPs in the sequel.

Definition 5 (2-step Recurrent). *Let $\mathcal{R}$ be an equivalence relation on S. A state s is said to be* 2-step recurrent *with respect to $\mathcal{R}$ iff s is not silent, $|Suc(s)| > 2$, and*

$$\exists(s \xrightarrow{\lambda} \mu).(\forall s' \in (Supp(\mu) \setminus [s]_{\mathcal{R}}).\forall(s' \xrightarrow{\lambda'} \nu).\nu(C) = 1), \qquad \text{(r1)}$$

where $C = ([s]_{\mathcal{R}} \cup [s']_{\mathcal{R}})$.

We say $\mathcal{C}$ is 2-step recurrent *with respect to $\mathcal{R}$, iff there exists $s \in S$ such that s is 2-step recurrent with respect to $\mathcal{R}$, otherwise it is non 2-step recurrent with respect to $\mathcal{R}$. Moreover, we say that s (or $\mathcal{C}$) is (non) 2-step recurrent iff it is (non) 2-step recurrent with respect to $\sim_{CSL}$.*

In other words, for a state s to be 2-step recurrent, it must be not silent and have more than 2 successors. Remind that each silent state can be replaced by a single state without changing properties of a CTMDP. After doing so, each silent state will only have one successor which is itself, so the requirement of non silence can be subsumed by $|Suc(s)| > 2$ in this case. Let us explain the more involved condition given in Eq. (r1). Eq. (r1) says that a 2-step recurrent state s must also satisfy: There exists $s \xrightarrow{\lambda} \mu$ such that for all states in $Supp(\mu)$ except those in $[s]_{\mathcal{R}}$, they can only evolve into states equivalent to s or themselves.

Example 5. We show some examples of (non) 2-step recurrent states. First of all, states s_0 and r_0 in Example 1 are 2-step recurrent, since they are not silent and have more than 2 successors. Moreover all successors u_i ($i = 1, 2, 3$) are silent, i.e., can only evolve into states which are CSL equivalent to themselves. However if we add an extra transition to u_3 as in Example 4, s_0 will be non 2-step recurrent, since u_3 can reach the state u_3' with probability 1, where u_3' is not CSL equivalent to either u_3 or s_0. For similar reasons, r_0 is also non 2-step recurrent.

Secondly, States s_1 and r_1 in Example 2 and s_2 and r_2 in Example 3 are trivially non 2-step recurrent, since the number of their successors is ≤ 2. □

Definition 5 seems tricky, however, we shall show that there exists an efficient scheme to check whether a given CTMDP is 2-step recurrent or not. More importantly, we shall see later in Remark 1 that the class of non 2-step recurrent CTMDPs contains an important part of CTMDP models, in particular those found in practice.

Now we are ready to show the main contribution of this paper. By restricting to the set of non 2-step recurrent CTMDPs, we are able to prove that the classical strong bisimulation defined in Definition 4 is both sound and complete with respect to the CSL equivalence, which is formalized in the following theorem.

Theorem 2. *If $\mathcal{C}$ is non 2-step recurrent, $\sim \;=\; \sim_{CSL}$.*

3.2 Weak Bisimulation

In this section we will introduce a novel notion of *weak bisimulation* for CTMDPs. Our definition of weak bisimulation is directly motivated by the well-known fact that uniformization does not alter time-bounded reachabilities for CTMDPs [27,30] when TTP schedulers are considered. Similar as in Section 3.1, we also show that weak bisimulation is both sound and complete for $\mathrm{CSL}_{\setminus \mathrm{X}}$ over non 2-step recurrent CTMDPs. We shall introduce the definition of weak bisimulation first.

Definition 6 (Weak bisimulation). *We say that states s and r in $\mathcal{C}$ are weak bisimilar, denoted by $s \approx r$, whenever $\bar{s} \sim \bar{r}$ in the uniformized* CTMDP $\bar{\mathcal{C}}$.

The way we define weak bisimulation here is different from the definition of weak bisimulation for CTMCs in [6], where a conditional measure is considered, see Definition 7 for the detailed definition. Moreover we will show in Section 5.2 that for CTMCs our weak bisimulation coincides with weak bisimulation defined in [6]. Even though the resulting uniformized CTMDP depends on the chosen rate E as shown in Definition 2, it is worth mentioning that weak bisimulation given in Definition 6 is independent of E. Since if two states are strong bisimilar in a uniformized CTMDP, they will be strong bisimilar in any uniformized CTMDP no matter which value we choose for E.

The following lemma establishes some properties:

Lemma 1

1. $\sim \; \subseteq \; \approx$,
2. *for uniformized* CTMDP*s*, $\sim \; = \; \approx$.

As we mentioned above, by uniformizing a CTMDP we will not change its satisfiability of $\mathrm{CSL}_{\setminus \mathrm{X}}$ provided that only TTP schedulers are considered. Therefore we have the following lemma saying that if two states satisfy the same formulas in $\mathrm{CSL}_{\setminus \mathrm{X}}$, then they will satisfy the same formulas in CSL after uniformization and vice versa.

Lemma 2. *$s \sim_{CSL_{\setminus \mathrm{X}}} r$ in $\mathcal{C}$ iff $\bar{s} \sim_{CSL} \bar{r}$ in $\bar{\mathcal{C}}$.*

The following theorem says that our weak bisimulation is sound for $\sim_{\mathrm{CSL}_{\setminus \mathrm{X}}}$, and particularly when the given CTMDP is non 2-step recurrent, weak bisimulation can be used to fully characterize $\mathrm{CSL}_{\setminus \mathrm{X}}$ equivalence.

Theorem 3. *$\approx \; \subsetneq \; \sim_{CSL_{\setminus \mathrm{X}}}$. If $\bar{\mathcal{C}}$ is non 2-step recurrent, $\approx \; = \; \sim_{CSL_{\setminus \mathrm{X}}}$.*

Theorem 3 works if we restrict to only TTP schedulers. However, this is not a restriction. Since it has been proved in [30,11] that there always exists an optimal scheduler in TTP for any path property in $\mathrm{CSL}_{\setminus \mathrm{X}}$.

3.3 Determining 2-step Recurrent CTMDPs

In Theorem 2 and 3, the completeness holds only for CTMDPs which are non 2-step recurrent. Hence it is important that 2-step recurrent CTMDPs can be checked efficiently. This section discusses a simple procedure for determining (non) 2-step recurrent CTMDPs. Before presenting the decision scheme, we shall introduce the following lemma, which holds by applying the definition of 2-step recurrent CTMDPs directly:

Lemma 3. *Given two equivalence relations $\mathcal{R}$ and $\mathcal{R}'$ over S such that $\mathcal{R} \subseteq \mathcal{R}'$, if $\mathcal{C}$ is 2-step recurrent with respect to $\mathcal{R}$, then it is 2-step recurrent with respect to $\mathcal{R}'$, or equivalently if $\mathcal{C}$ is non 2-step recurrent with respect to $\mathcal{R}'$, then it is non 2-step recurrent with respect to $\mathcal{R}$.*

Lemma 3 suggests a simple way to check whether a given CTMDP $\mathcal{C}$ is 2-step recurrent. Given an arbitrary equivalence relation $\mathcal{R}$ such that $\sim \;\subseteq\; \sim_{\mathrm{CSL}} \;\subseteq\; \mathcal{R}$, by Lemma 3, we can first check whether $\mathcal{C}$ is 2-step recurrent with respect to $\mathcal{R}$. Proper candidates for $\mathcal{R}$ should be as fine as possible, but also can be determined efficiently. For instance, we can let $\mathcal{R} = \{(s, r) \mid L(s) = L(r)\}$, or a finer equivalence relation defined as follows: $s\ \mathcal{R}\ r$ iff for each $C \in S/\mathcal{R}$ and $s \xrightarrow{\lambda} \mu$, there exists $r \xrightarrow{\lambda} \mu'$ such that $\mu'(C) \geq \mu(C)$. Such $\mathcal{R}$ is coarser than $\sim_{\mathrm{CSL}}$, and can be computed efficiently in polynomial time.

If $\mathcal{C}$ is not 2-step recurrent with respect to $\mathcal{R}$, we know that $\mathcal{C}$ is non 2-step recurrent with respect to $\sim_{\mathrm{CSL}}$ either. Otherwise we continue to check whether $\mathcal{C}$ is 2-step recurrent with respect to $\sim$, if the answer is yes, then $\mathcal{C}$ is 2-step recurrent with respect to $\sim_{\mathrm{CSL}}$ too. Note that $\sim$ can also be computed in polynomial time, see [38] for details. In the remaining cases, namely when $\mathcal{C}$ is 2-step recurrent with respect to $\mathcal{R}$, but not for $\sim$, we cannot conclude anything, instead the relation $\sim_{\mathrm{CSL}}$ shall be computed first for a definite answer.

As we discussed above, sometimes we need to use $\sim_{\mathrm{CSL}}$ to decide whether a given CTMDP is 2-step recurrent or not. But it turns out that $\sim_{\mathrm{CSL}}$ is hard to compute in general. Actually, we can prove the following lemma showing that the decision of $\sim_{\mathrm{CSL}}$ and $\sim_{\mathrm{CSL}_{\setminus \mathrm{X}}}$ is NP-hard.

Lemma 4. *It is NP-hard to decide whether $s \sim_{CSL} r$ and $s \sim_{CSL_{\setminus \mathrm{X}}} r$.*

Remark 1. We have implemented the above described scheme to check whether some models in practice are 2-step recurrent or not. Even though the implemented classification scheme is not complete since we do not compute CSL equivalence, it has been shown quite useful in practice. Our initial experiments show that the non 2-step recurrent CTMDPs consist of most models in practice. For instance the models of “Erlang Stages” [39], “Stochastic Job Scheduling” [10], “Fault-Tolerant Work Station Cluster” [19,23], and “European Train Control System” [7] are all non 2-step recurrent, which means that strong bisimulation coincides with $\sim_{\mathrm{CSL}}$ on these models. To be more confident, we also checked MDP models from the PRISM [25] benchmark interpreted as CTMDP models by interpreting all probabilities as rates. We found that all of them are non 2-step recurrent. □

4 Bisimilarity and CSL* Equivalence

In this section we study the relation between bisimilarity and CSL* equivalence. We first introduce CSL*, then show that strong bisimulation can be fully characterized by CSL* for arbitrary CTMDPs. Then we extend the work to weak bisimulation.

4.1 CSL*

As CTL* and PCTL* can be seen as extensions of CTL and PCTL respectively, CSL* can also be seen as an extension of CSL, where the path formula is defined by the Metric Temporal Logic (MTL) [24]. MTL extends linear temporal logic [29] by associating each temporal operator with a time interval. It is a popular logic used to specify properties of real-time systems and has been extensively studied in the literature [1,28,8,21]. The logic MTL was also extended to CTMCs in [13], where the authors studied the problem of model checking CTMCs against MTL specifications. Formally, the syntax of CSL* is defined by the following BNFs:

$$\begin{aligned}\varphi &::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{P}_{\bowtie p}(\psi),\\ \psi &::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}^I\,\psi \mid \psi \mathsf{U}^I \psi.\end{aligned}$$

The semantics of state formulas is the same as CSL, while the semantics of path formulas is more involved, since we may have different and embedded time bounds. As for MTL, there are two different semantics for the path formulas: continuous semantics and pointwise semantics. These two semantics make non-trivial differences in real-time systems, see [28] for details. We shall focus on the pointwise semantics as for CSL in this paper. Given a path ω and a path formula ψ of CSL*, the satisfiability $\omega \models \psi$ is defined inductively as follows: $\omega \models a$ iff $a \in L(\omega[0])$, $\omega \models \neg\psi$ iff $\omega \not\models \psi$, $\omega \models \psi_1 \wedge \psi_2$ iff $\omega \models \psi_1 \wedge \omega \models \psi_2$, $\omega \models \mathsf{X}^I\,\psi$ iff $\omega|_1 \models \psi \wedge \mathit{time}(\omega, 0) \in I$, and

$$\omega \models \psi_1 \mathsf{U}^I \psi_2 \text{ iff } \exists i.(\omega|_i \models \psi_2 \wedge \sum_{0 \le j < i} \mathit{time}(\omega, j) \in I \wedge (\forall 0 \le j < i.\omega|_j \models \psi_1)).$$

4.2 Strong Bisimulation

In this section we prove the soundness and completeness of strong bisimulation with respect to CSL* equivalence. Different from CTL and its extension CTL*, whose equivalences coincide on labelled transition systems [9], the extension from CSL to CSL* is non-trivial, as we shall show in this section that CSL* can fully characterize strong bisimulation for arbitrary CTMDPs. We reconsider Example 1 for inspiration:

Example 6. Let s_0 and r_0 be the states introduced in Example 1, where we have shown that s_0 and r_0 are not bisimilar, but satisfy the same CSL formula. However if we consider CSL*, s_0 and r_0 are not CSL* equivalent. It suffices to find a formula φ in CSL* such that $s_0 \models \varphi$, but $r_0 \not\models \varphi$. Let $\psi := (L(s_0)\mathsf{U}^{[0.6,\infty)}L(u_1)) \vee (L(s_0)\mathsf{U}^{[1,\infty)}L(u_3))$, then the maximal probability of paths starting from s_0 and satisfying ψ is equal to $\max\{0.3\cdot e^{-0.6}+0.4\cdot e^{-1}, 0.5\cdot e^{-0.6}+0.1\cdot e^{-1}\} < 0.312$, while the probability for r_0 is equal to $\max\{0.3\cdot e^{-0.6}+0.4\cdot e^{-1}, 0.4\cdot e^{-0.6}+0.3\cdot e^{-1}, 0.5\cdot e^{-0.6}+0.1\cdot e^{-1}\} > 0.312$, thus $s_0 \models \mathcal{P}_{\le 0.312}(\psi)$, while $r_0 \not\models \mathcal{P}_{\le 0.312}(\psi)$, which indicates $s_0 \not\sim_{\mathrm{CSL}^*} r_0$. Note ψ is not a valid formula in CSL, since it is the disjunction of two until operators. □

In the remainder of this section, we shall focus on the proof of $\sim \; = \; \sim_{\mathrm{CSL}^*}$. First, we introduce the following lemma in [32]:

Lemma 5 (Theorem 5 [32]). *Given a path formula ψ of* CSL* *and a state s, there exists a set of cylinder sets $Cyls$ such that $Sat(\psi) = \cup_{C \in Cyls} C$.*

As a direct result of Lemma 5, $Sat(\psi)$ is measurable for any path formula ψ of CSL*, as $Sat(\psi)$ can be represented by a countable set of measurable cylinders.

Now we are ready to present the main result of this section, i.e., strong bisimulation coincides with CSL* equivalence for arbitrary CTMDPs:

Theorem 4. *For any* CTMDP, $\sim \; = \; \sim_{CSL^*}$.

4.3 Weak Bisimulation

In this section we shall discuss the relation between weak bisimulation and the equivalence induced by $\mathrm{CSL}^*_{\backslash \mathsf{X}}$. Similar as in Section 4.2 for strong bisimulation, weak bisimulation can be fully characterized by $\mathrm{CSL}^*_{\backslash \mathsf{X}}$.

Since our weak bisimulation is defined as strong bisimulation on the uniformized CTMDPs, foremost we shall make sure that $\mathrm{CSL}^*_{\backslash \mathsf{X}}$ is preserved by uniformization under TTP schedulers, that is, we shall prove the following lemma:

Lemma 6. *$s \sim_{CSL^*_{\backslash \mathsf{X}}} r$ in $\mathcal{C}$ iff $\bar{s} \sim_{CSL^*} \bar{r}$ in $\bar{\mathcal{C}}$.*

As a side contribution, we extend the result in [27,30] and show that uniformization also does not change properties specified by $\mathrm{CSL}^*_{\backslash \mathsf{X}}$, provided TTP schedulers are considered. Given Lemma 6, the soundness and completeness of $\approx$ with respect to $\sim_{\mathrm{CSL}^*_{\backslash \mathsf{X}}}$ are then straightforward from Definition 6 and the fact that $\sim$ is both sound and complete with respect to CSL*.

Theorem 5. *For any* CTMDP, $\approx \; = \; \sim_{CSL^*_{\backslash \mathsf{X}}}$.

Currently, we only prove Theorem 5 with respect to TTP schedulers. However, the optimal scheduler for a CSL* formula may be not a TTP scheduler. Refer to the following example:

Example 7. Let $\mathcal{C}$ be a CTMDP as in Fig. 3, where the letter on above of each state denotes its label. Moreover states s_8 and s_9 only have self-loop transitions which are omitted. Let $\psi = ((a \vee b)\mathsf{U}^I d) \vee ((a \vee c)\mathsf{U}^I e)$ be a path formula of CSL*. We show that there exists a non-TTP scheduler π such that

$$Pr_{\pi,s_4}(\{\omega \in Paths^\infty(\mathcal{C}) \mid \omega \models \psi\}) > Pr_{\pi',s_4}(\{\omega \in Paths^\infty(\mathcal{C}) \mid \omega \models \psi\})$$

for any TTP scheduler π'. Let $I = [0, \infty]$. Since π' is a TTP scheduler, it can only make decision based on the elapsed time and the current state. When at s_7, π' will choose either the transition to s_8 or the transition to s_9 at each time point. Therefore the maximal probability of satisfying ψ is 0.5. However for a general scheduler π, it can make decision based on the full history. For instance when at s_7, we can let π choose the transition to s_8, if the previous state is s_5, otherwise s_9. Under this scheduler, the maximal probability of satisfying ψ is equal to 1, which cannot be obtained by any TTP scheduler. From this example, we can see that an optimal scheduler for a CSL* formula may make it decision based on the elapsed time as well as the states visited.

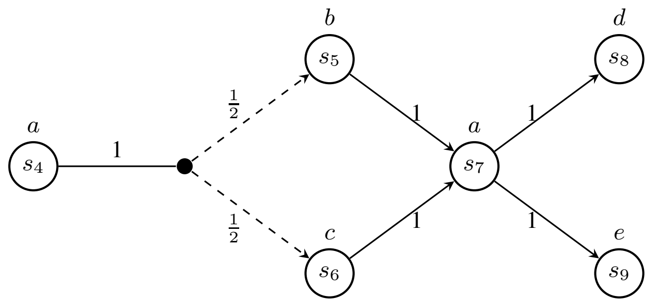


Fig. 3. TTP schedulers are not enough to obtain optimal values for CSL* properties

Example 7 shows that it is not enough to consider TTP schedulers in the setting of CSL*. In [27] another class of schedulers called *Total Time History dependent schedulers* (TTH) is introduced. We conjecture that for TTH schedulers: i) they preserve CSL$^*_{\backslash \mathsf{X}}$ properties after uniformization, and ii) they are powerful enough to obtain optimal values for CSL$^*_{\backslash \mathsf{X}}$ properties. Condition i) guarantees that Theorem 5 is valid, while condition ii) makes Theorem 5 general enough. We leave the proof of the conjecture as our future work.

Remark 2. The expressiveness of CSL* may be considered too powerful in certain cases. For instance, path formulas like $\Box(a\mathsf{U}^{[2,10]}b)$ [2] will be satisfied with probability 0 for any CTMDP. In general, if ψ can only be satisfied with probability strictly less than 1, the probability of satisfying ψ forever will be 0 for any CTMDP.

In the other hand, a small fragment of CSL* is enough to characterize strong bisimulation. Let CSL$^\vee$ denote the fragment of CSL* whose path formulas are defined by the following syntax: $\psi ::= \mathsf{X}^I\,\varphi \mid \psi \vee \psi$. We have shown in [34] that $\sim\; = \;\sim_{\mathrm{CSL}^\vee}$ for any CTMDP. Therefore any subset of CSL* which subsumes CSL$^\vee$ will be strong enough to fully characterize strong bisimulation.

5 Relation to MDPs and CTMCs

In this section, we compare related work on other stochastic models: MDPs and CTMCs.

5.1 Relation to (Weak) Bisimulation for MDPs

For MDPs, it is known that strong (probabilistic) bisimulation is only sound but not complete with respect to PCTL [31]–the counterpart of CSL in discrete setting. Differently, the completeness does not hold either even if we restrict to non 2-step recurrent MDPs, which can be defined in a straightforward way given Definition 5. Refer to the following example:

Example 8. Let s_0 and r_0 be two states as in Example 4, which will be viewed as two states in an MDP. Moreover we assume that u_3' only has a self loop. Since u_3' has

[2] $\Box\psi \equiv \neg((a \wedge \neg a)\mathsf{U}^{[0,\infty)}\neg\psi)$ for some a, i.e., ψ holds forever.

atomic propositions different from s_0 (r_0) and u_3, therefore s_0 and r_0 are not 2-step recurrent. However s_0 and r_0 satisfy the same PCTL formulas, since the maximal and minimal probabilities from s_0 and r_0 to any subset of $\{u_1, u_2, u_3, u_3'\}$ are the same. As mentioned before, the middle transition of r_0 cannot be simulated by any combined transition of s_0, hence they are not strong probabilistic bisimilar. This indicates that strong (probabilistic) bisimulation is not complete with respect to PCTL equivalence even that the given MDP is non 2-step recurrent. □

The counterpart of CSL* in discrete setting is PCTL* [3]. Similar as in the continuous case, the equivalence induced by PCTL* is strictly finer than $\sim_{\text{PCTL}}$ [33]. However, different from the continuous case, $\sim_{\text{PCTL}^*}$ is still coarser than strong (probabilistic) bisimulation for MDPs, that is, strong (probabilistic) bisimulation is not complete with respect to PCTL*:

Example 9. Let s_0 and r_0 be two states as in Example 1, where we have shown that s_0 and r_0 are neither strong bisimilar nor CSL* equivalent. However in [33] s_0 and r_0 are shown to be PCTL* equivalent by viewing them as two states in an MDP. Therefore CSL* gains more expressiveness by adding time bounds to the logic. □

The case for weak bisimulation is similar and omitted here.

5.2 Relation to (Weak) Bisimulation for CTMCs

In this section we show that our bisimulations are downward compatible to those for CTMCs. Different from CTMDPs, there is no non-deterministic transitions in CTMCs, i.e., each state has only one transition, which will be denoted by $s \xrightarrow{\lambda_s} \mu_s$. The notion of weak bisimulation can be found in [6] for CTMCs, which is repeated as follows:

Definition 7 (Weak Bisimulation of CTMCs). *For* CTMC*s, an equivalence relation $\mathcal{R}$ is a weak bisimulation iff for all $s \mathrel{\mathcal{R}} r$ it holds: i) $L(s) = L(r)$, and ii) $\lambda_s \cdot \mu_s(C) = \lambda_r \cdot \mu_r(C)$ for all equivalence classes $C \neq [s]_\mathcal{R}$.*

States s, r are weak bisimilar, denoted by $s \approx_{\mathit{CTMC}} r$, iff there exists a weak bisimulation $\mathcal{R}$ such that $s \mathrel{\mathcal{R}} r$.

Strong bisimulation for CTMCs is defined if in addition $\lambda_s \cdot \mu_s(C) = \lambda_r \cdot \mu_r(C)$ holds for $C = [s]_\mathcal{R} = [r]_\mathcal{R}$ as well. States s, r are strong bisimilar, denoted by $s \sim_{\text{CTMC}} r$, iff there exists a strong bisimulation $\mathcal{R}$ such that $s \mathrel{\mathcal{R}} r$.

Below we prove that, restricted to CTMCs, our strong and weak bisimulations agree with strong and weak bisimulations for CTMCs, respectively:

Lemma 7. *For CTMCs, it holds that $\sim \; = \; \sim_{\mathit{CTMC}}$ and $\approx \; = \; \approx_{\mathit{CTMC}}$.*

The lemma above shows that $\sim$ and $\approx$ are conservative extensions of strong and weak bisimulations for CTMCs in [6], and so are their logical characterization results except that they only work on a subset of CTMDPs free of 2-step recurrent states.

Since CTMCs are sub-models of CTMDPs,Theorem 4 and 5 also hold for CTMCs. Together with Lemma 7, we have the following result:

Corollary 1. *1. $\sim_{\mathit{CSL}^*} \; = \; \sim \; = \; \sim_{\mathit{CTMC}} \; = \; \sim_{\mathit{CSL}}$,*
2. $\sim_{\mathit{CSL}^*_{\setminus \mathrm{X}}} \; = \; \approx \; = \; \approx_{\mathit{CTMC}} \; = \; \sim_{\mathit{CSL}_{\setminus \mathrm{X}}}$.

Corollary 1 shows that CSL^* gains no more distinguishing power than CSL on CTMCs without non-determinism, similarly for their sub-logics without the next operator.

6 Conclusion and Future Work

In this paper, we have proposed both strong and weak bisimulations for CTMDPs, which are shown to be able to fully characterize CSL and $\mathrm{CSL}_{\setminus \mathsf{X}}$ equivalences respectively, but over non 2-step recurrent CTMDPs. For a standard extension of CSL – CSL^*, we show that strong and weak bisimulations are both sound and complete with respect to CSL^* and $\mathrm{CSL}^*_{\setminus \mathsf{X}}$ respectively for arbitrary CTMDPs. Moreover, we give a simple scheme to determine non 2-step recurrent CTMDPs, and show almost all CTMDPs found in practice are non 2-step recurrent CTMDPs. We note that the work in this paper can be extended to the simulation setting in a straightforward way.

For future work we would like to consider the approximation of bisimulations and simulations on CTMDPs as well as their logic characterization, along [15]. Moreover, the model checking of CSL^* against CTMCs and CTMDPs will be also worthwhile to exploit. Another interesting direction is to consider the continuous semantics of CSL^*.

Acknowledgements. Many thanks to the anonymous referees for their valuable suggestions on an early version of this paper. The authors are supported by IDEA4CPS and the VKR Center of Excellence MT-LAB. The work has received support from the EU FP7-ICT project MEALS (295261), and the DFG Sonderforschungsbereich AVACS. Part of the work was done while the first author was with IT University of Copenhagen, Denmark, and the second author was with Technical University of Denmark.

References

1. Alur, R., Henzinger, T.A.: A really temporal logic. J. ACM 41(1), 181–203 (1994)
2. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time Markov chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
3. Aziz, A., Singhal, V., Balarin, F., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: It usually works: The temporal logic of stochastic systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 155–165. Springer, Heidelberg (1995)
4. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. 29(6), 524–541 (2003)
5. Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. Theor. Comput. Sci. 345(1), 2–26 (2005)
6. Baier, C., Katoen, J.-P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for Markov chains. Inf. Comput. 200(2), 149–214 (2005)
7. Böde, E., Herbstritt, M., Hermanns, H., Johr, S., Peikenkamp, T., Pulungan, R., Wimmer, R., Becker, B.: Compositional performability evaluation for STATEMATE. In: QEST, pp. 167–178. IEEE (2006)
8. Bouyer, P., Markey, N., Ouaknine, J., Worrell, J.: The cost of punctuality. In: LICS, pp. 109–120. IEEE (2007)

9. Browne, M.C., Clarke, E.M., Grümberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comput. Sci. 59(1-2), 115–131 (1988)
10. Bruno, J., Downey, P., Frederickson, G.N.: Sequencing tasks with exponential service times to minimize the expected flow time or makespan. J. ACM 28(1), 100–113 (1981)
11. Buchholz, P., Hahn, E.M., Hermanns, H., Zhang, L.: Model checking algorithms for CTMDPs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 225–242. Springer, Heidelberg (2011)
12. Buchholz, P., Schulz, I.: Numerical analysis of continuous time Markov decision processes over finite horizons. Computers & Operations Research 38(3), 651–659 (2011)
13. Chen, T., Diciolla, M., Kwiatkowska, M., Mereacre, A.: Time-bounded verification of CTMCs against real-time specifications. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 26–42. Springer, Heidelberg (2011)
14. D'Argenio, P.R., Wolovick, N., Terraf, P.S., Celayes, P.: Nondeterministic labeled Markov processes: Bisimulations and logical characterization. In: QEST, pp. 11–20. IEEE (2009)
15. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov processes. Theor. Comput. Sci. 318(3), 323–354 (2004)
16. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Weak bisimulation is sound and complete for pCTL*. Inf. Comput. 208(2), 203–219 (2010)
17. Desharnais, J., Panangaden, P.: Continuous stochastic logic characterizes bisimulation of continuous-time Markov processes. J. Log. Algebr. Program. 56(1-2), 99–115 (2003)
18. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
19. Haverkort, B.R., Hermanns, H., Katoen, J.-P.: On the use of model checking techniques for dependability evaluation. In: SRDS, pp. 228–237 (2000)
20. Hermanns, H., Parma, A., Segala, R., Wachter, B., Zhang, L.: Probabilistic logical characterization. Inf. Comput. 209(2), 154–172 (2011)
21. Jenkins, M., Ouaknine, J., Rabinovich, A., Worrell, J.: Alternating timed automata over bounded time. In: LICS, pp. 60–69. IEEE (2010)
22. Jonsson, B., Larsen, K., Wang, Y.: Probabilistic extensions of process algebras. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 685–710. Elsevier (2001)
23. Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: QEST, pp. 167–176 (2009)
24. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
25. 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)
26. Neuhäußer, M.R., Katoen, J.-P.: Bisimulation and logical preservation for continuous-time Markov decision processes. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 412–427. Springer, Heidelberg (2007)
27. Neuhäußer, M.R., Stoelinga, M., Katoen, J.-P.: Delayed nondeterminism in continuous-time Markov decision processes. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 364–379. Springer, Heidelberg (2009)
28. Ouaknine, J., Worrell, J.: On the decidability of metric temporal logic. In: LICS, pp. 188–197. IEEE (2005)
29. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
30. Rabe, M.N., Schewe, S.: Finite optimal control for time-bounded reachability in CTMDPs and continuous-time Markov games. Acta. Inf. 48(5-6), 291–315 (2011)
31. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. Nord. J. Comput. 2(2), 250–273 (1995)

32. Sharma, A., Katoen, J.-P.: Weighted lumpability on Markov chains. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 322–339. Springer, Heidelberg (2012)
33. Song, L., Zhang, L., Godskesen, J.C.: Bisimulations meet PCTL equivalences for probabilistic automata. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 108–123. Springer, Heidelberg (2011)
34. Song, L., Zhang, L., Godskesen, J.C.: The branching time spectrum for continuous-time mdps. CoRR, abs/1204.1848 (2012)
35. van Glabbeek, R.J.: The linear time - branching time spectrum ii. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
36. van Glabbeek, R.J.: The linear time - branching time spectrum i. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 3–99. Elsevier (2001)
37. Wolovick, N., Johr, S.: A characterization of meaningful schedulers for continuous-time Markov decision processes. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 352–367. Springer, Heidelberg (2006)
38. Zhang, L., Hermanns, H., Eisenbrand, F., Jansen, D.N.: Flow faster: Efficient decision algorithms for probabilistic simulations. Logical Methods in Computer Science 4(4) (2008)
39. Zhang, L., Neuhäußer, M.R.: Model Checking Interactive Markov Chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)

Probabilistic Automata for Safety LTL Specifications

Dileep Kini and Mahesh Viswanathan

University of Illinois at Urbana-Champaign
Department of Computer Science

Abstract. Automata constructions for logical properties play an important role in the formal analysis of the system both statically and dynamically. In this paper, we present constructions of finite-state probabilistic monitors (FPM) for safety properties expressed in LTL. FPMs are probabilistic automata on infinite words that have a special, absorbing reject state, and given a cut-point $\lambda \in [0, 1]$, accept all words whose probability of reaching the reject state is at most $1 - \lambda$. We consider Safe-LTL, the collection of LTL formulas built using conjunction, disjunction, next, and release operators, and show that (a) for any formula φ, there is an FPM with cut-point 1 of exponential size that recognizes the models of φ, and (b) there is a family of Safe-LTL formulas, such that the smallest FPM with cut-point 0 for this family is of doubly exponential size. Next, we consider the fragment $\mathsf{LTL}(\mathbf{G})$ of Safe-LTL wherein always operator is used instead of release operator and show that for any formula $\varphi \in \mathsf{LTL}(\mathbf{G})$ (c) there is an FPM with cut-point 0 of exponential size for φ, and (d) there is a robust FPM of exponential size for φ, where a robust FPM is one in which the acceptance probability of any word is bounded away from the cut-point. We also show that these constructions are optimal.

1 Introduction

Connections between automata and logic have played an important role in providing mathematical foundations for understanding the tractability of logics and the notion of regularity. In addition, translations from logical properties to automata have been exploited to build efficient tools for model checking and monitoring of systems with respect to formal properties. In this paper we investigate the construction of probabilistic automata for safety properties expressed in Linear Temporal Logic.

Finite state probabilistic monitors (FPM), introduced in [1], are finite state automata on infinite strings that choose the next state on a transition based on the roll of a dice. They are a special class of Probabilistic Büchi automata (PBA) [2], wherein all states, except a special absorbing reject state, are accepting. Given a cut-point $\lambda \in [0, 1]$, an FPM accepts all words whose probability of reaching the reject state is at most $1 - \lambda$. We consider 3 special classes of FPMs in this paper. A *strong monitor* is an FPM with cut-point 1, i.e., an input word is accepted if the probability of reaching the reject state is 0. Thus a strong monitor never rejects a good word. A *weak monitor* is an FPM with cut-point 0, i.e., it accepts a word if it has some non-zero probability of being accepted. In other words, a weak monitor never accepts a bad word. Finally a *robust monitor* is an FPM (with cut-point λ) such that there is a gap $g > 0$ with the property that every input word is either accepted with probability at least $\lambda + g$ or at most $\lambda - g$. Robust monitors are automata with two-sided, bounded error. Therefore standard techniques like amplification can be used to have their error probabilities reduced. That is one can execute an input on many independent instances of the monitor and take the majority answer to have the error reduced by an exponential factor in the number of repetitions.

We investigate the construction of strong monitors, weak monitors, and robust monitors, for safety properties expressible in Linear Temporal Logic. Our motivations for this investigation are three-fold. Understanding the power of nondeterminism and randomization has driven research in theoretical computer science for the last 5 decades. The relationship between nondeterministic and deterministic models of finite automata is pretty well understood, both in terms of expressive power and efficiency. For probabilistic finite automata [1], while there have been many results characterizing their expressive power [3,4,2,1], there are almost no results studying their efficiency. Second, FPMs, as per their original motivation, model randomized algorithms that monitor system behavior at runtime. The construction of optimal FPMs for logical properties can yield efficient monitoring algorithms. Finally, model checking algorithms verifying Markov chains against LTL specifications exploit the translation of LTL formulas into deterministic automata. However, these translations can be both complicated and result in large automata. Instead, as argued in [5], probabilistic Büchi automata can be used when verifying qualitative LTL properties of Markov chains. Thus, translations from fragments of LTL to probabilistic automata can help in the efficient verification of Markov chains.

Safe-LTL are LTL properties built using propositions, conjunctions, disjunctions, next, and release operators; negations are restricted to being only applied to propositions. Safe-LTL is known to capture all safety properties expressible in LTL [6]. We show that for any property φ in Safe-LTL, there is a strong monitor with $O(2^{|\varphi|})$ states that accepts the models of φ. The monitor is essentially the nondeterministic Büchi automata for the property constructed in [7], where the nondeterministic choices are turned into probabilistic choices by assigning some non-zero probability to each choice. While this construction is not novel, it can be used to verify if a Markov chain violates a safety property with non-zero probability — since the PBA obtained by flipping the accept states and reject state of a FPM accepts the complement of the safety property with non-zero probability. We also prove that there is no translation from Safe-LTL to small weak monitors. More specifically, we show that there is a family of Safe-LTL properties $\{\varphi_n\}_{n\in\mathbb{N}}$ such that $|\varphi_n| = O(n \log n)$ and the smallest weak monitor for φ_n is of size $2^{2^{|\varphi|}}$.

[1] By probabilistic automata we loosely refer to both automata on finite and automata on infinite words.

Since all safety properties in LTL can be recognized by a doubly exponential deterministic automata, these results show that weak monitors maybe no smaller than deterministic automata. These results are surprising in the following light. Strong monitors are known to recognize only regular safety properties [1]. On the other hand, weak monitors are known to recognize all regular persistence properties [2] and even some non-regular properties [1]. Thus, while weak monitors recognize a richer class of properties than strong monitors, they are not as efficient.

Next, we consider a fragment of Safe-LTL, that we call LTL($\mathbf{G}$), where we only allow the use of the "always" modal operator instead of release. For this logic, we show that every formula φ has a weak monitor of size $O(2^{|\varphi|})$. Given results in [9], our construction demonstrates that weak monitors can be exponentially more succinct than deterministic automata for a large, natural class of properties. We also consider the construction of robust monitors for LTL($\mathbf{G}$). We show that for any property φ, there is a robust monitor of size $O(2^{|\varphi|})$ with gap $2^{-|\varphi|}$. Our construction is optimal in terms of the gap; we show that any robust monitor with gap $2^{-o(|\varphi|)}$ must be of size at least $2^{2^{\Omega(|\varphi|)}}$. Thus, robust monitors with subexponential gaps are no more efficient than deterministic automata for this logic. Our results are summarized in Figure 1.

	DBA	**NBA**	**Strong Monitors**	**Weak Monitors**	**Robust Monitors**
Safe-LTL	2-EXP	EXP	EXP	2-EXP	with gap $\frac{1}{2^{o(n)}}$: 2-EXP
LTL($\mathbf{G}$)	2-EXP	EXP	EXP	EXP	with gap $\frac{1}{2^{o(n)}}$: 2-EXP with gap $\frac{1}{2^n}$: EXP

Fig. 1. Size of automata recognizing Safety Properties in LTL. EXP means the number of states is exponential in the size of the formula and 2-EXP means that the number of states is doubly exponential. DBA stands for Deterministic Büchi Automata and NBA stands for Nondeterministic Büchi Automata.

We conclude the introduction with a brief discussion of our lower bound proofs which draw on results in communication complexity [10,11]. More specifically, we focus on *one round* protocols, where the only one message is sent from Alice to Bob, and Bob computes the value of the function based on this message and his input. We consider the *non-membership* problem, which is a special case of set disjointness problem, where Alice gets a subset $X \subseteq S$, and Bob gets $y \in S$, and they are required to determine if $y \in X$. We observe that the non-membership problem has a high VC-dimension, and, therefore, using results in [12], has a high communication complexity. Next, we observe that for certain languages, an FPM can be used to construct a one round protocol for this problem. In this protocol Alice and Bob construct special strings based on their inputs, Alice sends the state of the FPM after reading her input, and Bob computes the answer to the

[2] Informally, persistence properties are those that say that "eventually something always happens". For a formal definition see [8].

non-membership problem based on whether his string is accepted from the state sent by Alice. Thus, a lower bound on the communication complexity of the non-membership problem is lifted to obtain a lower bound on the number of states in an FPM recognizing certain languages.

The outline of the rest of the paper is as follows: In Section 2 we begin with prelimnaries, where we introduce FPMs, the safety fragment of LTL, and one round communication complexity. In Section 3 we present the constructions and lower bounds for the translating Safe-LTL specifications to strong and robust monitors. In Section 4 we present the results for going from the fragment $\mathsf{LTL}(\mathbf{G})$ to weak and robust monitors.

2 Preliminaries

First we introduce the notation that we will use throughout the paper. We use Σ to denote a finite set of symbols. σ to denote elements of Σ. Lower case greek letters $\alpha, \beta, \gamma \ldots$ will denote infinite strings from Σ^ω. We will use $u, v, w \ldots$ to denote finite strings in Σ^*. For any string $\alpha = \sigma_0\sigma_1 \ldots$ we will use $\alpha(i) = \sigma_i$ to denote the ith symbol in the sequence σ_i. The notation α_i will be used to denote the suffix begining from the ith position which is $\sigma_i\sigma_{i+1} \ldots$, and we use $\overleftarrow{\alpha_i}$ to denote the prefix ending at position i which is $\sigma_0\sigma_1 \ldots \sigma_i$. For a finite set S a distribution over S is any function $\mu : S \to [0, 1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. Since S is finite μ can be thought of as a vector with $|S|$ co-ordinates. The set of all distributions over S will be denoted by $dist(S)$. A stochastic matrix δ is a square matrix with non-negative entries such that each row of the matrix sums up-to one.

2.1 Finite State Probabilistic Monitors

Definition 1. *A finite state probabilistic monitor* (FPM) $\mathcal{M}$ *is a tuple* $(Q, \Sigma, \mu_0, q_r, (\delta_\sigma)_{\sigma \in \Sigma})$ *where* Q *is a finite set of states;* $\mu_0 \in dist(Q)$ *is the initial distribution;* $q_r \in Q$ *is the reject state, and* $(\delta_\sigma)_{\sigma \in \Sigma}$ *is an indexed set of stochastic matrices with dimension* $|Q| \times |Q|$ *such that* $\delta_\sigma(q_r, q_r) = 1$ *for all* $\sigma \in \Sigma$.

We will use $|\mathcal{M}|$ *to denote the size of* Q. *For* $q \in Q$ *we will use* $(\mathcal{M}, q)$ *to denote the* FPM $(Q, \Sigma, \mu_0', q_r, (\delta_\sigma)_{\sigma \in \Sigma})$ *where* $\mu_0'(q) = 1$.

Given an infinite string $\alpha = \sigma_0\sigma_1\sigma_2 \ldots$ as input the FPM $\mathcal{M}$ behaves in the following manner. $\mathcal{M}$ first chooses a state q_1 according to μ_0. If the next symbol that it is reading is σ then it moves to state q_2 with probability $\delta_\sigma(q_1, q_2)$; consumes the input symbol σ and keeps repeating this process for the remaining input.

For a finite word $u = \sigma_0 \ldots \sigma_n$ define δ_u as the matrix product $\delta_{\sigma_0} \ldots \delta_{\sigma_n}$, and let $\mu_{\mathcal{M},u}$ be the distribution $\mu_0\delta_u$. Define the rejection probability of u as $\mu^{rej}_{\mathcal{M},u} = \mu_{\mathcal{M},u}(q_r)$ and and acceptance probability $\mu^{acc}_{\mathcal{M},u} = 1 - \mu^{rej}_{\mathcal{M},u}$. Next observe that $\mu^{rej}_{\mathcal{M},\overleftarrow{\alpha_i}} \leq \mu^{rej}_{\mathcal{M},\overleftarrow{\alpha_{i+1}}}$ because the reject state has no edges leaving it. So, the sequence $\mu^{rej}_{\mathcal{M},\overleftarrow{\alpha_0}}\mu^{rej}_{\mathcal{M},\overleftarrow{\alpha_1}} \ldots$ is non decreasing and since it is upper-bounded by 1, its limit exists. Define $\mathcal{M}$'s *probability of rejecting* α, denoted by $\mu^{rej}_{\mathcal{M},\alpha}$, to be this limit. Let $\mathcal{M}$'s *probability of accepting* α be $\mu^{acc}_{\mathcal{M},\alpha} = 1 - \mu^{rej}_{\mathcal{M},\alpha}$.

Given a cutpoint $\lambda \in [0,1]$, the language recognized by $\mathcal{M}$ can be defined as the set of all infinite words with probability of acceptance at least λ or strictly more than λ. This is formally defined below.

Definition 2. *Given a cut-point* $\lambda \in [0,1]$ *and FPM* $\mathcal{M}$

- $L_{>\lambda}(\mathcal{M}) = \{\alpha \in \Sigma^\omega \mid \mu^{acc}_{\mathcal{M},\alpha} > \lambda\}$
- $L_{\geq\lambda}(\mathcal{M}) = \{\alpha \in \Sigma^\omega \mid \mu^{acc}_{\mathcal{M},\alpha} \geq \lambda\}$

Example: Figure 2 shows an example of an FPM. Here the input alphabet consists of Boolean assignments for p and q that is $\Sigma = 2^{\{p,q\}}$. The transitions are shown to be annotated with predicates over p and q: for example the transition from 1 to 2 is annotated with $p, \frac{1}{2}$ which means if you are in state 1 and see an input symbol σ for which $p \in \sigma$ then you move to state 2 with probability $\frac{1}{2}$. Consider the initial distribution to be the one which assigns equal probabilities to states $\{1,2,3\}$ and let q_r be the reject state. The language associated with $\mathcal{M}$ at cutpoints 1 and 0 are: $L_{\geq 1}(\mathcal{M}) = [\![\mathbf{G}(p \wedge q)]\!]$ and $L_{>0} = [\![\mathbf{G}(p \vee \mathbf{G}q)]\!]$ (The temporal logic and its semantics are defined in the next section).

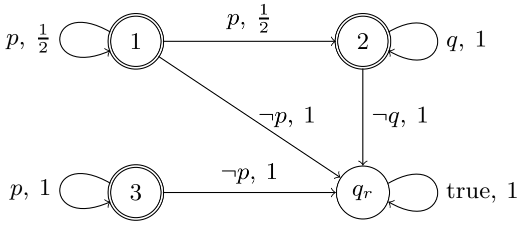


Fig. 2. Example FPM $\mathcal{M}$

A cut-point is said to be extremal if it is either 0 or 1, and it is called non-extremal otherwise. The following proposition states that when the cutpoint is non-extermal it does not matter as to what that exact cutpoint is. The proposition is proved in [1] and a proof sketch is included here because the same construction will be used in Corollary 2.

Proposition 1. *[1] For any* $p \in [0,1]$ *and for any* FPM $\mathcal{M}$

- *there is an* FPM $\mathcal{M}'$ *of size* $O(|\mathcal{M}|)$ *such that* $\mu^{rej}_{\mathcal{M}',\alpha} = p\mu^{rej}_{\mathcal{M},\alpha}$
- *there is an* FPM $\mathcal{M}'$ *of size* $O(|\mathcal{M}|)$ *such that* $\mu^{acc}_{\mathcal{M}',\alpha} = p\mu^{acc}_{\mathcal{M},\alpha}$.

Proof. Let $\mathcal{M}$ be $(Q, \Sigma, \mu_0, q_r, (\delta_\sigma)_{\sigma\in\Sigma})$. First let us consider when there is a single state q_0 for which $\mu_0(q_0) = 1$. In this case we introduce two new states $q_0', q_{acc} (\notin Q)$ and extend the transition matrices to include $\delta_\sigma(q_0', q) := p\delta_\sigma(q_0, q)$, $\delta_\sigma(q_0', q_{acc}) := (1-p)$, $\delta_\sigma(q_{acc}, q_{acc}) := 1$ and $\delta_\sigma(q, q_0') := \delta_\sigma(q_0', q_0') = 0$ for all

$q \in Q$ and $\sigma \in \Sigma$. Let $\mathcal{M}'$ be $(Q \cup \{q'_0, q_{acc}\}, \Sigma, \mu'_0, q_r, (\delta_\sigma)_{\sigma\in\Sigma})$ where $\mu'_0(q'_0) := 1$. The construction ensures that every word is diverted to q_{acc} right at the beginning with probability $(1-p)$. A word α gets rejected on $\mathcal{M}'$ if it does not go to q_{acc} and ends up in q_r. If it does not go to q_{acc} then the word would behave exactly as it would on $\mathcal{M}$ except for the beginning, where the probability gets scaled by p, and hence $\mu^{rej}_{\mathcal{M}',\alpha} = p\mu^{rej}_{\mathcal{M},\alpha}$. When there are multiple states $q \in Q$ with $\mu_0(q) > 0$ one can introduce a copy for each of them in the same way.

The second part can be similarly proved by diverting probabilities to q_r instead of q_{acc} at the beginning.

Corollary 1. *For any two cutpoints* $\lambda_1, \lambda_2 \in (0,1)$ *and* FPM $\mathcal{M}_1$*, there is a* FPM $\mathcal{M}_2$ *such that* $L_{>\lambda_1}(\mathcal{M}_1) = L_{>\lambda_2}(\mathcal{M}_2)$ *and* $L_{\geq\lambda_1}(\mathcal{M}_1) = L_{\geq\lambda_2}(\mathcal{M}_2)$ *and* $\mathcal{M}_2$ *is of size* $O(|\mathcal{M}_1|)$.

We will be interested in FPMs for which the probability of accepting any word is bounded away from the cut-point. We call these FPMs to be robust, and we define this formally. Robustness, first introduced in [1], is analogous to the concept of isolated cut-points [3] for probabilistic automata over finite words.

Definition 3. *Given* FPM $\mathcal{M}$ *and a cut-point* λ *define* $gap(\mathcal{M}, \lambda)$ *as*

$$gap(\mathcal{M}, \lambda) = \inf_{\alpha\in\Sigma^\omega} |\mu^{acc}_{\mathcal{M},\alpha} - \lambda|$$

Definition 4. *A* FPM *is said to be* ***robust*** *with respect to a cut-point* λ *if* $gap(\mathcal{M}, \lambda) > 0$. *When a* $\mathcal{M}$ *is robust for cut-point* λ *then* $L_{>\lambda}(\mathcal{M}) = L_{\geq\lambda}(\mathcal{M})$ *which we will represent by* $L_\lambda(\mathcal{M})$.

For robust FPMs, we can always consider the cut-point to be $\frac{1}{2}$ without seriously changing the size of the automata or its gap. Thus in the rest of the paper, we will assume that the cut-point of any robust monitor that we consider is $\frac{1}{2}$.

Corollary 2. *If* FPM $\mathcal{M}$ *is robust at* λ *then there is a* FPM $\mathcal{M}'$ *of the same size as* $\mathcal{M}$ *such that* $L_\lambda(\mathcal{M}) = L_{\frac{1}{2}}(\mathcal{M}')$ *and* $gap(\mathcal{M}', \frac{1}{2}) \geq \frac{1}{2} gap(\mathcal{M}, \lambda)$

Proof. The construction of Proposition 1 give us the required FPM $\mathcal{M}'$. When $\lambda < \frac{1}{2}$ choose $p := \frac{1}{2(1-\lambda)}$ and observe that:

$$gap(\mathcal{M}', \tfrac{1}{2}) = \inf_{\alpha\in\Sigma^\omega} |\mu^{acc}_{\mathcal{M}',\alpha} - \tfrac{1}{2}| = \inf_{\alpha\in\Sigma^\omega} |\tfrac{1}{2} - \mu^{rej}_{\mathcal{M}',\alpha}|$$

$$= \inf_{\alpha\in\Sigma^\omega} |\tfrac{1}{2} - \tfrac{\mu^{rej}_{\mathcal{M},\alpha}}{2(1-\lambda)}| = \frac{\inf_{\alpha\in\Sigma^\omega} |\mu^{acc}_{\mathcal{M},\alpha} - \lambda|}{2(1-\lambda)} \geq \tfrac{1}{2} gap(\mathcal{M}, \lambda)$$

Monitorable Languages. An FPM $\mathcal{M}$ is said to *strongly monitor* a language $L \subseteq \Sigma^\omega$ if $L = L_{\geq 1}(\mathcal{M})$, meaning the monitor $\mathcal{M}$ never declares a correct behaviour to be wrong. Similarly $\mathcal{M}$ is said to *weakly monitor* L if $L_{>0}(\mathcal{M}) = L$ which means $\mathcal{M}$ never accepts a wrong behaviour but it may occasionally reject a correct behaviour. The FPM $\mathcal{M}$ is said to *robustly monitor* L if there is a cut-point λ for which $\mathcal{M}$ is robust and $L = L_\lambda(\mathcal{M})$.

Gap Amplification. Given a robust FPM $\mathcal{M}$, one can always increase the gap (and reduce the error probability) by running multiple copies of $\mathcal{M}$ in parallel. Before presenting this result, we introduce some formal definitions that will allow us to state this result precisely.

A family of FPMs is a sequence of FPMs $\{\mathcal{M}_n\}_{n\in\mathbb{N}}$. We define the size of the family as the function $s(n) = |\mathcal{M}_n|$, and the gap of the family as the function $g(n) = gap(\mathcal{M}_n, \frac{1}{2})$.

Lemma 1. *Let $\{\mathcal{M}_n\}$ be a family of robust* FPM*s, then there exists a family $\{\mathcal{M}'_n\}$ of size $s(n)^{\lceil\frac{1}{g(n)^2}\rceil}$ with $\Omega(1)$ gap such that for all n $L_{\frac{1}{2}}(\mathcal{M}_n) = L_{\frac{1}{2}}(\mathcal{M}'_n)$.*

Proof. Gap amplification is well known technique wherein a particular experiment is repeated in order to increase the gap or reduce the error probability [13]. Consider $\mathcal{M}'_n$ to be the machine that runs $\lceil\frac{1}{g(n)^2}\rceil$ copies of $\mathcal{M}_n$ in parallel on an input word α. $\mathcal{M}'_n$ rejects α if more than $\frac{1}{2}$ of the $\lceil\frac{1}{g(n)^2}\rceil$ copies of $\mathcal{M}_n$ reject α. Using Chernoff's bounds we can show that for any α, $|\mu^{acc}_{\mathcal{M}'_n,\alpha} - \frac{1}{2}| \geq \frac{1}{2} - e^{-2}$. The bounds on the size and the gap for $\mathcal{M}'_n$ follow from these observations.

2.2 Safety Specifications

In this section, we introduce fragments of LTL that describe safety properties, and for which we will present constructions of FPMs.

Definition 5. **(Safe-LTL syntax)** *Given a finite set of propositions P the formulae in* Safe-LTL *fragment of linear temporal logic over P is given by the following grammar:*

$$\varphi ::= p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\,\mathbf{R}\,\varphi$$

Definition 6. **(Safe-LTL semantics)** *Formulae in* Safe-LTL *over the set of propositions P are interpreted over words in $(2^P)^\omega$ which are sequences of assignments to propositions. The semantics of the logic is given by the following rules:*

$$\begin{array}{ll}
\alpha \vDash p & \textit{iff } p \in \alpha(i) \\
\alpha \vDash \neg p & \textit{iff } p \notin \alpha(i) \\
\alpha \vDash \varphi_1 \wedge \varphi_2 & \textit{iff } \alpha \vDash \varphi_1 \textit{ and } \alpha \vDash \varphi_2 \\
\alpha \vDash \varphi_1 \vee \varphi_2 & \textit{iff } \alpha \vDash \varphi_1 \textit{ or } \alpha \vDash \varphi_2 \\
\alpha \vDash \mathbf{X}\varphi & \textit{iff } \alpha_1 \vDash \varphi \\
\alpha \vDash \varphi_1\,\mathbf{R}\,\varphi_2 & \textit{iff either } \forall j \in \mathbb{N} : \alpha_j \vDash \varphi_2 \\
& \quad \textit{or } \exists k \in \mathbb{N} : (\alpha_k \vDash \varphi_1 \textit{ and } \forall j \leq k \;\; \alpha_j \vDash \varphi_2)
\end{array}$$

The semantics of φ denoted by $[\![\varphi]\!]$, is given by $[\![\varphi]\!] = \{\alpha \in (2^P)^\omega \mid \alpha \vDash \varphi\}$

Definition 7. *The logic* $\mathsf{LTL}(\mathbf{G})$ *over set of propositions* P *is the fragment of* $\mathsf{Safe\text{-}LTL}$ *over* P *where the allowed temporal operators are* $\mathbf{X}$ *(next) and* $\mathbf{G}$ *(always). It is given by the grammar:*

$$\varphi ::= p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \mathbf{G}\varphi$$

where p *ranges over* P*. Since* $\mathbf{G}\varphi$ *will be interpreted as* $\mathbf{false}\,\mathbf{R}\,\varphi$*,* $\mathsf{LTL}(\mathbf{G})$ *is a fragment of* $\mathsf{Safe\text{-}LTL}$*. The semantics of* $\mathbf{G}$ *can be given as:* $\alpha \vDash \mathbf{G}\varphi$ *iff* $\forall i \in \mathbb{N} : \alpha_i \vDash \varphi$

From the semantics of $\mathbf{X}$, one can infer the following equivalences:

$$\mathbf{X}(\varphi_1 \vee \varphi_2) \equiv \mathbf{X}\varphi_1 \vee \mathbf{X}\varphi_2 \qquad \mathbf{X}(\varphi_1\,\mathbf{R}\,\varphi_2) \equiv (\mathbf{X}\varphi_1)\,\mathbf{R}\,(\mathbf{X}\varphi_2)$$
$$\mathbf{X}(\varphi_1 \wedge \varphi_2) \equiv \mathbf{X}\varphi_1 \wedge \mathbf{X}\varphi_2 \qquad \mathbf{X}\mathbf{G}\varphi \equiv \mathbf{G}\mathbf{X}\varphi$$

Therefore for any formula in $\mathsf{Safe\text{-}LTL}$ or $\mathsf{LTL}(\mathbf{G})$ the next operator can be pushed inside the scope of any other operator. This results in a quadratic blow up in the size of the formula, but the size of the formula apart from the $\mathbf{X}$s will remain the same. We will exploit this observation in the constructions ahead.

2.3 Automata over Infinite Words

A *nondeterministic Büchi automaton* (NBA) is a tuple $(Q, \Sigma, Q_0, \delta, F)$ where Q is a finite set of states; $Q_0 \subseteq Q$ is the initial set of states; Σ is a finite set of input symbols; $\delta : Q \times \Sigma \to 2^Q$ is the transition function and $F \subseteq Q$ is the set of good states. A run of a NBA N over a word $\alpha \in \Sigma^\omega$ is an infinite sequence of states $\rho = q_0 q_1 q_2 \cdots \in Q^\omega$ such that $q_0 \in Q_0$ and for each $i \in \mathbb{N}$ $q_{i+1} \in \delta(q_i, \alpha(i))$. A run is called accepting if there are infinitely many indices i for which $q_i \in F$. A word $\alpha \in \Sigma^\omega$ is said to be accepted by N if α has an accepting run. $L(B)$ is defined as the set of all infinite words over Σ which are accepted by N.

Lemma 2. *[14] For every formula* φ *in* $\mathsf{Safe\text{-}LTL}$ *there is an NBA* $N = (Q, \Sigma, Q_0, \delta, \{q\})$ *such that* $\delta(q, \sigma) = \{q\}$ *for all* $\sigma \in \Sigma$*,* $[\![\neg\varphi]\!] = L(N)$ *and size of* N *is* $O(2^{|\varphi|})$*.*

Proof. Follows from the constructions of Theorem 22 and Proposition 20 from [14]. The NBA constructed from the alternating automaton for $\neg\varphi$ has a single final state with outgoing edges only to itself.

2.4 Communication Complexity

In the two-party communication complexity model of Yao [10], there are two parties Alice and Bob, who are given strings $x \in X$ and $y \in Y$, respectively, and are trying to cooperatively compute a Boolean function $f : X \times Y \to \{0, 1\}$ on their joint inputs. We are primarily interested in one-round communication protocols, wherein Alice computes some (randomized) function a on her input x, sends the output of this function to Bob, and then Bob, based on Alice's message and his own input, tries to compute the answer $f(x, y)$ using another function b. This can be formalized as below.

Definition 8. *A one round protocol is $P = (a, Z, b)$, where $a : X \times R_A \to Z$ is the (randomized) function that Alice computes (where R_A is the space of Alice's random choices), Z is the space of messages that Alice uses to communicate to Bob, and $b : Z \times Y \times R_B \to \{0, 1\}$ is the (randomized) function that Bob computes (where again R_B is the space of Bob's random choices).*

- *The protocol P is said to* compute f with error at most $\epsilon \in (0, 1)$ *if*

$$\Pr_{r_1 \in_R R_A,\, r_2 \in_R R_B} \big[b(a(x, r_1), y, r_2) = f(x, y)\big] \geq 1 - \epsilon$$

  where the probability is measured over the random choices made by Alice and Bob.
- *The* cost *of protocol P will be taken to be the number of bits communicated by Alice to Bob in the worst case, i.e.,* $\mathsf{cost}(P) = \log |Z|$. *Notice, that in measuring the cost of the protocol, we do not measure the resources needed to compute the functions a and b.*
- *The* randomized one-round communication complexity *of function f, denoted by $R^{A\to B}_{\epsilon}(f)$, is the least cost of any one-round protocol computing f with error at most ϵ.*

The one-round communication compexity of a boolean function $f(x, y)$ is closely related to the concept of VC-dimension. We take a look at the definition VC-dimension [15].

Definition 9. *Let H be a class of boolean functions over the set Y. A set $Y' \subseteq Y$ is said to be* shattered *by H if for every $T \subseteq Y'$ there exists a function $h_T \in H$ such that for all $y \in Y'$ $h_T(y) = 1$ iff $y \in T$. The size of the largest set $Y' \subseteq Y$ which is shattered by H is known as the* VC-dimension *of H and is denoted by $VC\text{-}dim(H)$.*

For a Boolean function $f : X \times Y \to \{0, 1\}$ let $f_x : Y \to \{0, 1\}$ be the function defined as $f_x(y) = f(x, y)$ and let $f_X = \{f_x \mid x \in X\}$.

Lemma 3. *[12] For any boolean function $f : X \times Y \to \{0, 1\}$ over finite sets X and Y, and any constant error $\epsilon < \frac{1}{8}$, the one-round randomized communication complexity of f, $R^{A\to B}_{\epsilon}(f) = \Omega(VC\text{-}dim(f_X))$.*

We will be interested in a particular function which we will call the *non membership function.* Given any set S the non-membership function is the Boolean function $g^S : X \times Y \to \{0, 1\}$ where $X = 2^S$, $Y = S$ and $g(x, y)$ is 1 when $y \notin x$ and 0 otherwise.

Proposition 2. *The VC-dimension of the class of functions g^S_X is equal to the size of S, that is $VC\text{-}dim(g^S_X) = |S|$.*

Proof. For every subset $Y' \subseteq S$ the function $g^S_x \in g^S_X$ where $x = S - Y' \in X$ is such that $g^S_x(y) = 1$ iff $y \in Y'$.

Corollary 3. *For error $\epsilon < \frac{1}{8}$, the one-round randomized communication complexity of the non-membership function $R^{A\to B}_{\epsilon}(g^S)$ is $\Omega(|S|)$*

3 Monitors for Safe-LTL

In this section, we present a construction of strong monitors for formulae in Safe-LTL of exponential size, and also show that the smallest weak monitors for some formulas in Safe-LTL can be doubly exponential sized. However, before presenting these results, we observe that any exponential blow-up is inevitable even for very simple formulas that are built using only the **X** operator. Thus the best upper bounds we can hope for is exponential.

Proposition 3. – *There exists a family of specification $\{\varphi_n\}_{n\in\mathbb{N}}$ such that any family $\{\mathcal{M}_n\}_{n\in\mathbb{N}}$ that weakly monitors it has size at least $2^{|\varphi_n|}$, and*
– *There exists a family of specfication $\{\varphi_n\}_{n\in\mathbb{N}}$ such that any family $\{\mathcal{M}_n\}_{n\in\mathbb{N}}$ that strongly monitors it has size at least $2^{|\varphi_n|}$*

Proof. Consider the class of languages $\{L_n\}$ where $L_n = \{uu(0+1)^\omega \mid u \in \{0,1\}^n\}$. L_n can be specified by saying that for each $i \in \{1,\ldots,n\}$ the ith input symbol should be the same as the $(i+n)$th input symbol, which can be done using only **X** and the boolean connectives. Any FPM that weakly recognizes L_n should have at least 2^n states because for each $u \in \{0,1\}^n$ one can identify a state q reachable from u and not reachable on any other $v \in \{0,1\}^n$ such that the word u is accepted with non-zero probability from q.

The complement of the above L_n serves to show that translation to strong monitors need also result in an exponential blowup, and can be argued in a similar fashion.

3.1 Strong Monitors

We present our construction of an exponential size strong monitor for Safe-LTL formulas.

Theorem 1. *For every formula φ in* Safe-LTL *there is a* FPM *$\mathcal{M}_\varphi$ of size $O(2^{|\varphi|})$ such that $\mathcal{M}_\varphi$ strongly monitors $[\![\varphi]\!]$ that is $L_{\geq 1}(\mathcal{M}_\varphi) = [\![\varphi]\!]$*

Proof. We begin by using Lemma 2 to construct a NBA $B = (Q, \Sigma, Q_0, \delta, q_f)$ that recognizes all words that don't satisfy the specification φ, such that state q_f is absorbing. Let $\mu_0 \in dist(Q)$ such that $\mu_0(q) = \frac{1}{|Q_0|}$ if $q \in Q_0$ and 0 otherwise. For all $q_1, q_2 \in Q$ define $\delta_\sigma(q_1, q_2) = \frac{1}{|\delta(q_1,\sigma)|}$ if $q_2 \in \delta(q_1, \sigma)$ and 0 otherwise. The required FPM $\mathcal{M}_\varphi$ is $(Q, \Sigma, \mu_0, q_f, (\delta_\sigma)_{\sigma\in\Sigma})$.

Any word that satisfies φ can never reach the reject state of the FPM, if it could then the accept state of B would also be reachable on that word and since it is absorbing the word would be accepted by B, which is not possible as B accepts words that do not satisfy φ. Hence no word satisfying φ can reach the reject state of $\mathcal{M}_\varphi$ so we have $[\![\varphi]\!] \subseteq L_{\geq 1}(\mathcal{M}_\varphi)$. Any word that does not satisfy φ can reach the accepting state of the NBA, and hence can reach the reject state in $\mathcal{M}_\varphi$ with non-zero probability and so we have that $[\![\varphi]\!] \supseteq L_{=1}(\mathcal{M}_\varphi)$.

The above result is not a novel construction. However, it can be potentially exploited in model checking Markov Chains. Flipping the accept and reject states of the FPM gives a PBA that accepts the complement of the safety property with non-zero probability, hence the above construction shows that there are exponential sized PBAs recognizing co-safety properties expressed in LTL. Thus, if to check if a Markov chain violates a safety property with non-zero probability, we can use this PBA instead of constructing a deterministic automaton for the co-safety property (which can be doubly exponential in size).

3.2 Weak Monitors

While Safe-LTL admits strong monitors of exponential size, we show that the smallest weak monitors for some formulas can be doubly exponential in size. This is interesting in the light of the fact that weak monitors are expressively more powerful than strong monitors (i.e., can recognize languages not recognizable by strong monitors) [1].

Theorem 2. *There exists a family of* Safe-LTL *specification* $\{\varphi_n\}_{n\in\mathbb{N}}$ *of size* $O(n \log n)$ *such that any family of* FPM*s that weakly monitors* $\{\varphi_n\}_{n\in\mathbb{N}}$ *has size* $2^{\Omega(2^n)}$

Proof. Consider $\Sigma = \{0, 1, \#, \$\}$ and the following ω languages over Σ

$$\begin{aligned} S_n &= (\# \cdot (0+1)^n)^+ \cdot \$ \cdot (0+1)^n \\ R'_n &= \{(\# \cdot (0+1)^n)^* \cdot (\# \cdot w) \cdot (\# \cdot (0+1)^n)^* \cdot \$ \cdot w \mid w \in (0+1)^n\} \\ R_n &= S_n - R'_n \\ L_n &= R_n^\omega + R_n^* \cdot (\# \cdot (0+1)^n)^\omega \end{aligned}$$

A word in S_n can be thought of as an instance of a non-membership query: call the set of n-bit strings appearing before the \$ as the query set and a n-bit string appearing after the \$ as the query string. In a non-membership query you want to know whether the query string does not occur in the query set. R'_n represents the words in S_n that are *no* instances of the non-membership query and R_n represents the *yes* instances. L_n represents either a possibly infinite sequences of yes instances of the non-membership query or finitely many yes instances followed by an infinite sequence of n-block $0, 1$ separated by # alone. For the Safe-LTL specification the set of propositions P we consider is $\{0, 1, \#, \$\}$, and we will assume that exactly one proposition holds at any time point. This can be easily enforced by a constant sized specification. For the sake of simplicity we present the specification family that is of size $O(n^2)$. For details about the more succinct $O(n \log n)$ specification the interested reader should refer to [16] after reading this proof.

$$\# \wedge \mathbf{G}(\# \Rightarrow (\bigwedge_{i=1}^{n} \mathbf{X}^i(0 \vee 1)) \wedge \mathbf{X}^{n+1}(\# \vee \$)) \tag{1}$$

$$\wedge\, \mathbf{G}(\$ \Rightarrow (\bigwedge_{i=1}^{n} \mathbf{X}^i(0 \vee 1)) \wedge \mathbf{X}^{n+1}(\#)) \tag{2}$$

$$\wedge\, \mathbf{G}(\# \Rightarrow \bigvee_{i=1}^{n} \bigvee_{\sigma \in \{0,1\}} (\mathbf{X}^i(\sigma) \wedge (\mathbf{X}(\$ \wedge \mathbf{X}^i(\sigma^c))\, \mathbf{R}\, (\neg \$))) \tag{3}$$

(1) says that the words should begin with # and each # should be followed by a n-bit string followed by a # or \$. (2) says that every \$ is followed by n-bit string block then followed by #. (1) and (2) together describe a sequence of possibly infinite non-membership queries. (3) says that for any every word in the query set differs from the query string in at least one position. Hence (1) $\wedge$ (2) $\wedge$ (3) is a specification of L_n and is of size $O(n^2)$.

Now let us assume that we have a family of FPMs $\{\mathcal{M}_n\}$ that weakly monitors $\{L_n\}$. We make the following claim about $\mathcal{M}_n$.

Claim. Consider any $n \in \mathbb{N}$ and $c \in (0,1)$. Let $\mathcal{M}_n$ be $(Q, \Sigma, Q_0, q_r, (\delta_\sigma)_{\sigma \in \Sigma})$. There exists a $u \in R_n^*$ and a state $q \neq q_r$ with $\mu_{\mathcal{M}_n,u}(q) > 0$ such that for all $\beta \in R_n^\omega : \mu^{acc}_{(\mathcal{M}_n,q),\beta} \geq c$.

Proof. Suppose the claim does not hold for some c and n. Let us fix $\mathcal{M}$ to be $\mathcal{M}_n$. Since the claim is false we have that for any $u \in R_n^*$ and any q with $\mu_{\mathcal{M},u}(q) > 0$ there is a $\beta_q \in R_n^\omega$ such that the measure of accepting runs from q on β_q is less than c. Let us fix q to be a state with maximal $\mu_{\mathcal{M},u}(q)$ among all $q \neq q_r$. For such a q we have $\mu_{\mathcal{M},u}(q) \geq \frac{\mu^{acc}_{\mathcal{M},u}}{|\mathcal{M}|}$ by pigeon-hole principle. For an FPM the acceptance measure of any string is non-increasing along its length and so there should be a finite prefix of β_q, say $v \in R_n^*$ such that $\mu^{acc}_{(\mathcal{M},q),v} < c$. From this we get

$$\mu^{acc}_{\mathcal{M},uv} \leq \mu^{acc}_{\mathcal{M},u}\left(1 - \frac{1-c}{|\mathcal{M}|}\right)$$

because at least $1-c$ of the probability of reaching q is lost to q_r after seeing v. So for any $u \in R_n^*$ we manage to find a string $v \in R_n^*$ such that the acceptance probability of the extended string uv compared to u decreases by a constant factor. But observe that uv is once again in R_n^*. So this extension process can be repeated forever to get a string in R_n^ω which is accepted with 0 probability which is a contradiction. So our claim is indeed true.

Consider $S = \{0,1\}^n$. In order to show the required lower bound on $\mathcal{M}_n$ we will show how $\mathcal{M}_n$ can be used to construct a constant-error one-round protocol for the non-membership function g^S. For the sake of the remaining argument we instantiate c in the above claim to $\frac{7}{8}$. Consider the state q as per our claim. Let η denote the string $(\#a^n)^\omega$. We make the following observations:

– for $w \in R_n$ the acceptance probability of $w\eta$ starting from q is at least $\frac{7}{8}$. This is because from q every prefix w' of $w\eta$ should be accepted with probability at least $\frac{7}{8}$, otherwise we can attach an appropriate suffix to w' to get a string that contradicts our claim.

– for $w \in R'_n$, the string $w\eta$ is accepted with 0 probability because $\mathcal{M}_n$ should accept strings $uw\eta$ with 0 probability.

So $\mathcal{M}_n$ if started from q is able to significantly distinguish between *yes* and *no* instances of the non-membership query. We can use this to construct a one-round protocol for the function g^S: Alice encodes her input set of n-bit blocks as a string in $(\# \cdot (0+1)^n)^+\$$ and runs it on $\mathcal{M}_n$ starting from q and sends to Bob the resulting state q'. Bob then simulates $y\eta$ from q' and outputs 0 iff the simulation results in rejection. (Bob cannot actually run the infinite string η but he can simulate η's acceptance because probability of η being accepted from any state can be calculated). This gives us a randomized one-round protocol with error $< \frac{1}{8}$. The number of bits exchanged in this protocol is $\log_2 |\mathcal{M}|$, but according to Corollary 3 any such protocol needs to exchange at least $\Omega(2^n)$ bits. Hence we get that $\mathcal{M}$ has at least $2^{\Omega(2^n)}$ states.

4 Monitors for LTL(G)

The results in Section 3 show that for general Safe-LTL formulas, weak monitors can be as large as deterministic automata. In this section, we show that when we consider the sub-logic LTL($\mathbf{G}$), we can demonstrate that weak monitors can be exponentially more succinct than their deterministic counterparts. The idea behind the construction is that we consider each state to represent a guess about the truth of all subformulae of the form $\mathbf{G}\psi$: whether it holds now, or holds starting from some point in the future, or never holds. Then we argue that for satisfying behaviours one can find accepting runs that need to make finitely many correct guesses and vice versa.

In the latter part of the section we present constructions of robust monitors for this logic that are small. This result relies on considering a normalized form of the formula which yields an efficient way to construct the required monitor.

4.1 Weak Monitors

Before we present our construction of exponential sized weak monitors for the fragment LTL($\mathbf{G}$) , we introduce some assumptions and definitions that will facilitate the proof.

For a formula φ let $Sub(\varphi)$ denote the set of all subformulae of φ, and let $GSub(\varphi) \subseteq Sub(\varphi)$ be those which are of the form $\mathbf{G}\psi$.

Definition 10. *A **annotation** for a formula φ is a function mapping $GSub(\varphi)$ to the set $\{\top, \bot, \mathsf{L}\}$. Denote by A the set of all annotations. A annotation is called **stable** if it maps $GSub(\varphi)$ to $\{\top, \bot\}$. Given a annotation a and $\sigma \in 2^P$ an **evaluation** for φ is the unique function $e_a^\sigma : Sub(\varphi) \to \{\top, \bot\}$ that meets the following constraints:*

$e^\sigma_a(\psi) \;=\top$ iff $(a(\psi) = \top)$ for $\psi \in GSub(\varphi)$
$e^\sigma_a(p) \;=\top$ iff $p \in \sigma$ $\qquad e^\sigma_a(\psi_1 \wedge \psi_2)=e^\sigma_a(\psi_1) \wedge e^\sigma_a(\psi_2)$
$e^\sigma_a(\neg p)=\top$ iff $p \notin \sigma$ $\qquad e^\sigma_a(\psi_1 \vee \psi_2)=e^\sigma_a(\psi_1) \vee e^\sigma_a(\psi_2)$

A annotation represents a guess for each subformula $\mathbf{G}\psi$ stating whether $\mathbf{G}\psi$ holds now ($\top$), holds for some later point but not now (L), never holds ($\bot$). An evaluation attempts to evaluate the truth for all the subformulae (read $e^\sigma_a(\psi)$ as evaluation of ψ annotated with a and σ). Note that an evaluation need not be logically consistent. For example $e^\sigma_a(\mathbf{G}p)$ could be $\top$ because $a(\mathbf{G}p) = \top$, but $e^\sigma_a(p) = \bot$ because $p \notin \sigma$. We are now ready to present the main result of this section.

Theorem 3. *For every $\varphi \in \mathsf{LTL}(\mathbf{G})$ there is a FPM $\mathcal{M}_\varphi$ of size $2^{O(|\varphi|)}$ such that $L_{>0}(\mathcal{M}_\varphi) = [\![\varphi]\!]$*

Proof. First we show how the construction works for $\varphi \in \mathsf{LTL}(\mathbf{G})$ when it does not have any $\mathbf{X}$ operators. Let Q the set of states be $(A\times\{0,1\})\cup\{q_r\}$. For $\sigma \in 2^P$ define T_σ to be the binary relation on Q such that for $a, b \in A$, $((a, t_1), (b, t_2)) \in T_\sigma$ iff: $t_2 = 0$ and if $t_1 = 1$ then $e^\sigma_a(\varphi) = \top$ and the following conditions on a, b holds:

$$(a(\mathbf{G}\psi) = \top) \Rightarrow (b(\mathbf{G}\psi) = \top \wedge e^\sigma_a(\psi) = \top)$$
$$(a(\mathbf{G}\psi) = \bot) \Rightarrow (b(\mathbf{G}\psi) = \bot)$$
$$(a(\mathbf{G}\psi) = \mathsf{L}) \Rightarrow (b(\mathbf{G}\psi) \neq \bot)$$

For $a \in A$, $((a, t_1), q_r) \in T_\sigma$ if for no (b, t_2) the above conditions hold. For all σ, $(q_r, q_r) \in T_\sigma$. Define δ_σ as:

$$\delta_\sigma(q_1, q_2) = \frac{T_\sigma(q_1, q_2)}{\sum_{q_2} T_\sigma(q_1, q_2)},$$

Define μ_0 as follows: for an annotation a, $\mu_0((a, t)) = \frac{1}{|A|}$ if $t = 1$ and define it to be 0 on the rest of the states. The required FPM $\mathcal{M}_\varphi$ is $(Q, 2^P, \mu_0, q_r, (\delta_\sigma)_{\sigma\in\Sigma})$. Now we prove that $L_{>0}(\mathcal{M}_\varphi) = [\![\varphi]\!]$.

$L_{>0}(\mathcal{M}_\varphi) \supseteq [\![\phi]\!]$: For a string α that satisfies φ we look at the sequence of states induced by α, i.e define the ith state (a_i, t_i) as $t_i = 1$ iff $i = 0$ and

$$a_i(G\psi) = \begin{cases} \top & \text{if } \alpha_i \models \mathbf{G}\psi \\ \mathsf{L} & \text{if } \alpha_i \not\models \mathbf{G}\psi \text{ and } \exists j > i : \alpha_j \models \mathbf{G}\psi \\ \bot & \text{if } \forall j \geq i : \alpha_j \not\models \mathbf{G}\psi \end{cases}$$

First let us observe that $\delta_{\alpha(i)}((a_i, t_i), (a_{i+1}, t_{i+1})) > 0$ for any i: If $a_i(\mathbf{G}\psi) = \top$ then by construction, $\alpha_i \models \mathbf{G}\psi$. This implies $\alpha_{i+1} \models \mathbf{G}\psi$ and so $a_{i+1}(\mathbf{G}\psi) = \top$. Also one can prove that for all $i \in \mathbb{N}$ and $\psi \in Sub(\varphi)$, $(\alpha_i \models \psi) \Rightarrow e^{\alpha(i)}_{a_i}(\psi) = \top$ by induction on the structure of ψ . If $a_i(\mathbf{G}\psi) = \mathsf{L}$ then we know ψ is going to hold forever from some point after i, which means it is also going to hold forever from some point after $i + 1$ and hence $a_{i+1}(\mathbf{G}\psi) \neq \bot$. If $a_i(\mathbf{G}\psi) = \bot$

then we know ψ is going to be false infinitely often from i, so ψ will be false infinitely often from $i+1$ as well hence $a_{i+1}(\mathbf{G}\psi) = \bot$. This makes sure that a_i and a_{i+1} are properly related. Since $\alpha \vDash \varphi$ we have that $e_{a_0}^{\alpha(0)}(\varphi) = 1$. Thus $((a_0, 1), (a_1, 0)) \in T_{\alpha(0)}$. Hence, $(a_0, 1)(a_1, 0) \ldots$ is a valid run of $\mathcal{M}_\varphi$ over α.

For any a_i if $\mathbf{G}\psi \in GSub(\varphi)$ is marked L then there is a $j > i$ such that $\mathbf{G}\psi \vDash \alpha_j$ (defnition of a_i), so it follows that $a_j(\mathbf{G}\psi) = \top$. This implies that every L eventually becomes $\top$, so there exists a point k at which a_k marks all formulae in $GSub(\varphi)$ as either $\top$ or $\bot$ (which cannot be further modified), and hence $a_{k'} = a_k$ for all $k' \geq k$. Once you reach a_k you cannot go to any other annotation except itself and hence for all $k' \geq k : \delta_{\alpha(k')}((a_k, 0), (a_k, 0)) = 1$. Therefore this run does not have to make any probabilistic choice after the point k and hence has positive acceptance probability.

$L_{>0}(\mathcal{M}_\varphi) \subseteq [\![\varphi]\!]$: Consider any valid accepting run $(a_0, t_0)(a_1, t_1) \ldots$ of $\mathcal{M}_\varphi$ on input α. First observe that in any valid run of $\mathcal{M}_\varphi$ the number of $\top$s and $\bot$s are non decreasing. Since there are finitely many states, the run ultimately stagnates at a particular state. Let us denote by C_a the set of runs which stay in the state $(a, 0)$ after finitely many steps. Denote by C_a^i the set of all runs in C_a that stay in a after i steps. We have $C_a = \cup_{i=1}^{\infty} C_a^i$.

Fix a to be an unstable annotation. For any σ we have $\delta_\sigma((a, 0), (a, 0)) < 1$, because a has a choice to change a L to $\top$ and move to a different annotation. So the probabiltiy measure associated with C_a^i is 0 because after i steps the only transition taken is from $(a, 0)$ to $(a, 0)$ which leaks at least $1 - \delta_\sigma((a, 0), (a, 0))$ probability out of $(a, 0)$. This implies the probability associated with the set C_a is also zero.

So if the set of all accepting runs of α has non-zero measure then it has to have a run that ultimately reaches a stable annotation. Now with such a run we prove that the word α satisfies φ. For any $\varphi' \in Sub(\varphi)$ and $i \in \mathbb{N}$ if $e_{a_i}^{\alpha(i)}(\varphi') = \top$ then $\alpha_i \vDash \varphi'$, this can be proved by performing induction on the structure of φ'. The interesting case is when $\varphi' = \mathbf{G}\psi$. The definition of e suggests that if $e_{a_i}^{\alpha(i)}(\mathbf{G}\psi) = \top$ then $a_i(\mathbf{G}\psi) = \top$. Since $((a_i, t_i), (a_{i+1}, t_{i+1})) \in T_{\alpha(i)}$, we get from the definition of T that $e_{a_i}^{\alpha(i)}(\psi) = \top$ and so it follows that $\alpha_i \vDash \psi$ from the induction hypothesis. But if $\mathbf{G}\psi$ is marked $\top$ in a_i then it is marked $\top$ in every a_j for $j > i$. So $\alpha_j \vDash \psi$ for every $j \geq i$ and hence we have that $\alpha_i \vDash \mathbf{G}\psi$. Finally observe that $((a_0, t_0), (a_1, t_1)) \in T_{\alpha(0)}$ iff $e_{a_0}^{\alpha(0)}(\varphi) = \top$. Thus $\alpha_0 \vDash \varphi$.

What remains is to be shown is the construction in the presence of $\mathbf{X}$ operators. First we push down the $\mathbf{X}$ operators to the bottom as we saw in Section 2.2. If the number of nested $\mathbf{X}$s is at most k (which is at most $|\varphi|$) then by looking ahead k positions into the input one can evaluate the $\mathbf{X}$ subformulae just like literals. So by maintaining the last k input symbols and delaying the computation by those many steps will give use the required construction. Since we need to remember k input symbols the construction will blow up only by a factor of $|\Sigma|^k$ which is again in $2^{|\varphi|}$.

4.2 Robust Monitors

We will present a construction of robust automata of exponential size for formulae in $\mathsf{LTL}(\mathbf{G})$. We begin by observing that the above construction for weak monitors does not result in a robust monitor. Consider the formula $\mathbf{G}(p \vee \mathbf{G}q)$. The construction of Theorem 3 results in the FPM given in Figure 3. The initial distribution gives equal probability to the states A, B and C. The states are as follows: $A = (a_1, 1)$, $B = (a_2, 1)$, $C = (a_3, 1)$, $A' = (a_1, 0)$, $B' = (a_2, 0)$ and $C' = (a_3, 0)$, where a_1, a_2, a_3 are annotations as given below.

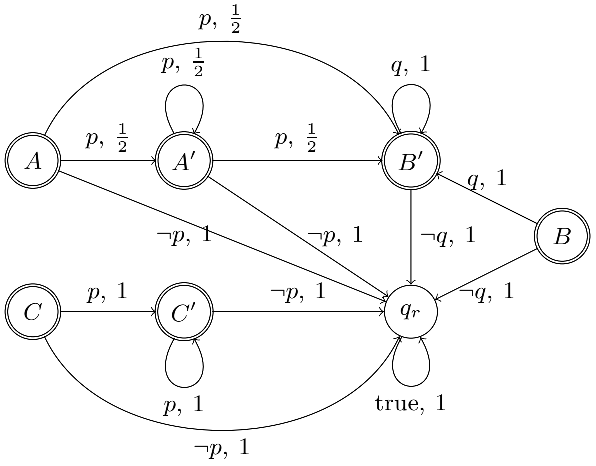


Fig. 3. Weak monitor for $\mathbf{G}(p \vee \mathbf{G}(q))$

$$\{a_1(\mathbf{G}(p \vee \mathbf{G}q)) = \top, a_1(\mathbf{G}q) = \mathsf{L}\} \quad \{a_2(\mathbf{G}(p \vee \mathbf{G}q)) = \top, a_2(\mathbf{G}q) = \top\}$$
$$\{a_3(\mathbf{G}(p \vee \mathbf{G}q)) = \top, a_3(\mathbf{G}q) = \bot\}$$

The rest of the annotations do not appear as they are unreachable. To see why the FPM is not robust we consider the word $p^n q^\omega$. After seeing the first $n > 0$ input symbols of this word, the monitor is going to be in state A' with probabitlity $\frac{1}{3.2^n}$, in state B' with $\frac{1}{3.2^n}$ and in state C' with probability $\frac{1}{3}$. Probability of being in state C' goes to 0 as we see the rest of string q^ω. This means as n grows larger the word $p^n q^\omega$ is accepted with negligible probability and hence the language $[\![\mathbf{G}(p \vee \mathbf{G}q)]\!]$ is not robustly monitored by this FPM. Therefore, we present a new construction that avoids these pitfalls.

Theorem 4. *For every φ in* $\mathsf{LTL}(\mathbf{G})$ *there is a* FPM $\mathcal{M}_\varphi$*, which is robust with $\frac{1}{2^{|\varphi|}}$ gap, $2^{O(|\varphi|)}$ states such that $\mathcal{M}_\varphi$ recognizes φ.*

Proof. As in Theorem 3 we can push all the $\mathbf{X}$ in the formula to the bottom and take care of it by remembering the last k input symbols where k is the nesting depth of the $\mathbf{X}$s. Therefore we are going to consider only formulae without any $\mathbf{X}$ in this proof.

The FPM that we construct is going to accept safe inputs with probability 1 and reject bad inputs with probability $> \frac{1}{2^{|\varphi|}}$.

Let us first consider the simpler logic $\mathsf{LTL}_{\vee}(\mathbf{G})$ built using literals, disjunction and $\mathbf{G}$; so the formulas have no conjunction. We will say that a formula $\varphi \in \mathsf{LTL}_{\vee}(\mathbf{G})$ is *guarded* iff every subformula $\mathbf{G}\psi$ of φ is of the form $\mathbf{G}(\alpha \vee \beta)$, where α is a disjunction of literals, and β is a disjunction of formulae like $\mathbf{G}\gamma$; β could be an empty disjunction. Observe that every formula φ in $\mathsf{LTL}_{\vee}(\mathbf{G})$ is equivalent to a guarded formula ψ such that $|\psi| = O(|\varphi|)$. This is because we have $\mathbf{G}\mathbf{G}\psi \equiv \mathbf{G}\psi$, and $\mathbf{G}(\mathbf{G}\alpha_1 \vee \mathbf{G}\alpha_2 \vee \cdots \vee \mathbf{G}\alpha_n) \equiv (\mathbf{G}\alpha_1 \vee \mathbf{G}\alpha_2 \vee \cdots \vee \mathbf{G}\alpha_n)$. Every guarded formula φ can be recognized by a deterministic monitor [3] of size $2^{|\varphi|}$, whose states keep track of the guarded subformulae which are yet to be violated. For example, if the formula is $\mathbf{G}(\alpha_1 \vee \mathbf{G}\alpha_2 \vee \mathbf{G}\alpha_3)$ then the automaton monitors α_1 until it becomes false and then starts monitoring $\mathbf{G}\alpha_2$ and $\mathbf{G}\alpha_3$ (which are guarded).

Consider $\varphi \in \mathsf{LTL}(\mathbf{G})$. Since $\wedge$ distributes over $\vee$ and $\mathbf{G}(\psi_1 \wedge \psi_2) \equiv (\mathbf{G}\psi_1) \wedge (\mathbf{G}\psi_2)$, we can pull all the conjunctions out, and show that φ is equivalent to a formula ψ which is conjunction of formulas in $\mathsf{LTL}_{\vee}(\mathbf{G})$. We can also see that $|\psi| = O(2^{|\varphi|})$. The FPM for ψ will be will be a disjoint union of the deterministic monitors recognizing each of the conjuncts in ψ. Thus, the number of states in this FPM is thus $O(2^{2|\varphi|})$. The initial distribution assigns equal probability to the initial states of each of the deterministic monitors (and 0 probability to all other states). A bad input violates one of the conjuncts, and so the monitor corresponding to that conjunct will reject the input. Thus, bad inputs are accepted with probability at most $(1 - \frac{1}{2^{|\varphi|}})$. On the other hand, a good input is accepted by the monitors of each of the conjuncts, and so is accepted by the FPM for ψ with probability 1.

The proof of Theorem 4 constructs a robust FPM with exponential gap. We show that these bounds on the gap cannot be improved without increasing the number of states to doubly exponential; this is the content of the next theorem.

Theorem 5. *There exists a family of* $\mathsf{LTL}(\mathbf{G})$ *(and hence* $\mathsf{Safe\text{-}LTL}$*) specifications* $\{\varphi_n\}_{n\in\mathbb{N}}$ *of size* $O(n)$ *s.t. any family of robust* FPM*s with* $\frac{1}{2^{o(n)}}$ *gap that recognizes it has size* $2^{2^{\Omega(n)}}$.

Proof. We consider the specfications used in [9] to prove lower bounds on deterministic generators. Let $\varphi_n = \mathbf{G}(\bigvee_{i=1}^{n}(\neg p_i \wedge \mathbf{G}\neg q_i))$. We will reduce the problem of finding efficient protocols for the non-membership function to the problem of finding small sized FPMs for this specification.

Let $P_p = \{p_1, \ldots, p_n\}$, $P_q = \{q_1, \ldots, q_n\}$, Σ_k be $\{a \subseteq P_p \mid |a| = k\}$ and Γ_k be $\{b \subseteq P_q \mid |b| = k\}$. We choose k to be $\frac{n}{2}$ so that $|\Sigma_k| = 2^{\Omega(n)}$. For any $\sigma \subseteq P_p$ let $q(\sigma) = \{q_i \mid p_i \notin \sigma\}$. Let $S = \Sigma_k$ and g^S be the corresponding non-membership

[3] A deterministic monitor is an FPM in which each transition matrix has entries which are either 0 or 1.

function. Using Corollary 3 we get that for $\epsilon \leq \frac{1}{8}$ the communication complexity $R_\epsilon^{A\to B}(g^S) \in 2^{\Omega(n)}$.

We begin by showing that a constant gap family $\{\mathcal{M}_n\}$ recognizing $\{\varphi_n\}$ should have large size. Consider an FPM family $\mathcal{M}_n$ with gap at least $\frac{3}{8}$ such that $L(\mathcal{M}_n) = [\![\varphi_n]\!]$. Now we are going construct a randomized one-round protocol for g^S with $< \frac{1}{8}$ error using $\mathcal{M}_n$. Alice encodes her input x as a string $\sigma_1\sigma_2\ldots\sigma_m$, an enumeration of the sets in x (which are also symbols in the alphabet $2^{P_p \cup P_q}$), runs it on $\mathcal{M}_n$ and gives the resulting state to Bob. Bob whose input is y simulates the word $q(y)(\emptyset)^\omega$ from the given state, and outputs 0 if it results in rejection and outputs 1 otherwise. A word violates φ_n iff there is a point in the word where for each p_i that is false it is the case that eventually q_i is true. Suppose x and y are such that $g^S(x,y) = 0$, this means that there is some $\sigma_j \in x$ for which $y = \sigma_j$ and so we get that $\sigma_1\sigma_2\ldots\sigma_m q(y)(\emptyset)^\omega$ violates φ_n. Similarly if $g^S(x,y) = 1$ then is no j such that $\sigma_j \in x$ and $y = \sigma_j$ and so $\sigma_1\sigma_2\ldots\sigma_m q(y)(\emptyset)^\omega$ satisfies φ_n. Since $\mathcal{M}_n$ has a gap of $\frac{3}{8}$ it follows that the protocol that we constructed has an error of at most $\frac{1}{8}$ in deciding the output $g^S(x,y)$. The number of bits exchanged in this protocol is $\log_2|\mathcal{M}_n|$. But above we saw that any such protocol should exchange $2^{\Omega(n)}$ bits which imples that the size of $\mathcal{M}_n$ has to be $2^{2^{\Omega(n)}}$. Having shown that constant gap FPMs recognizing $\{\varphi_n\}$ is of size $2^{2^{\Omega(n)}}$ we invoke Lemma 1 to get the same lower bound for $\frac{1}{2^{o(n)}}$ gap FPMs.

5 Conclusion

In this paper we gave constructions of FPMs for safety properties expressed in LTL. We showed that Safe-LTL has strong monitors of exponential size, where as weak monitors and robust monitors with sub-exponential gaps, can be doubly exponentially large. For the sub-logic LTL($\mathbf{G}$) we gave constructions of weak monitors and robust monitors of exponential size. However, the gap for robust monitors for LTL($\mathbf{G}$) given by our construction is exponential and we showed that these bounds on the gap cannot be improved without increasing the number of states to doubly exponential.

A number of questions remain open. While we showed that robust monitor with sub-exponential gap for Safe-LTL can be doubly exponential in size, we could not conclude if the construction can be improved if we relax the bounds on the gap. In particular, it would be interesting to know if there are exponential sized robust monitors with an exponential gap. More generally, our results say nothing about properties beyond safety, and good probabilistic Büchi automata constructions for general LTL could have practical applications in verification.

Acknowledgements. Dileep Kini was supported by NSF grant CCF-1016989. Mahesh Viswanathan was supported by NSF grant CNS-1016791.

References

1. Chadha, R., Sistla, A.P., Viswanathan, M.: On the expressiveness and complexity of randomization in finite state monitors. J. ACM 56(5), 26:1–26:44 (2009)
2. Baier, C., Gröβer, M.: Recognizing ω-regular languages with probabilistic automata. In: Proceedings of the IEEE Symposium on Logic in Computer Science, pp. 137–146 (2005)
3. Rabin, M.: Probabilitic automata. Information and Control 6(3), 230–245 (1963)
4. Paz, A.: Introduction to Probabilistic Automata. Academic Press (1971)
5. Baier, C., Bertrand, N., Größer, M.: On decision problems for probabilistic büchi automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 287–301. Springer, Heidelberg (2008)
6. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspect of Computing, 495–511 (1999)
7. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999)
8. Manna, Z., Pnueli, A.: Temporal verification of reactive and concurrent systems: Specification. Springer (1992)
9. Alur, R., La Torre, S.: Deterministic generators and games for ltl fragments. ACM Trans. Comput. Logic 5(1), 1–25 (2004)
10. Yao, A.: Some complexity questions related to distributed computing. In: Proceedings of the ACM Symposium on Theory of Computation, pp. 209–213 (1979)
11. Kushilevtiz, E., Nisan, N.: Communication Complexity. Cambridge University Press (1996)
12. Kremer, I., Nisan, N., Ron, D.: On randomized one-round communication complexity. In: Symposium on Theory of Computing (June 1995)
13. Motwani, R., Raghavan, P.: Randomized algorithms. Cambridge University Press, New York (1995)
14. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
15. Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability & Its Applications 16(2), 264–280 (1971)
16. Kupferman, O., Rosenberg, A.: The blow-up in translating LTL to deterministic automata. In: van der Meyden, R., Smaus, J.-G. (eds.) MoChArt 2010. LNCS, vol. 6572, pp. 85–94. Springer, Heidelberg (2011)

Refuting Heap Reachability

Bor-Yuh Evan Chang

University of Colorado Boulder
bec@cs.colorado.edu

Abstract. Precise heap reachability information is a prerequisite for many static verification clients. However, the typical scenario is that the available heap information, computed by say an up-front points-to analysis, is not precise enough for the client of interest. This imprecise heap information in turn leads to a deluge of false alarms for the tool user to triage. Our position is to approach the false alarm problem not just by improving the up-front analysis but by also employing after-the-fact, goal-directed *refutation analyses* that yield targeted precision improvements. We have investigated refutation analysis in the context of detecting statically a class of Android memory leaks. For this client, we have found the necessity for an overall analysis capable of path-sensitive reasoning interprocedurally and with strong updates—a level of precision difficult to achieve globally in an up-front manner. Instead, our approach uses a refutation analysis that mixes highly precise, goal-directed reasoning with facts derived from the up-front analysis to prove alarms false and thus enabling effective and sound filtering of the overall list of alarms.

The depth and breadth of what static verification tools can do is really quite astounding. However, the unfortunate truth is that for a software developer to fully realize the benefits of static verification, she must go through the onerous task of triaging the warnings that such a tool produces to determine whether the warnings are true bugs or false alarms. Only then can bug fixes be brought to bear. The situation is particularly dramatic when we consider the verification of modern programs that make extensive use of heap allocation. Precise heap information is a prerequisite for effective reasoning about nearly any non-trivial property of such programs. While there is a large body of work on heap reasoning, including in broad areas like pointer analysis and shape analysis, the all-too-common situation is that the available heap information is not quite precise enough for the client of interest.

Our position is to approach the false alarm problem not just by improving up-front precision but also with analyses for alarm triage that can wield after-the-fact, targeted precision improvements on demand. In this context, we investigate the combination of a scalable, off-the-shelf points-to analysis as an up-front analyzer with an ultra-precise, after-the-fact refutation analysis for heap reachability queries. The particular challenge that we examine is how to maximally leverage the combination of the up-front analysis and the after-the-fact refutation analysis.

The development of our approach has been driven in part by building a tool for detecting a pernicious class of memory leaks in Android applications. In brief, such a leak occurs when an operating system object of type `Activity` is reachable from a static field after the end of its lifecycle. At this point, the `Activity` object is no

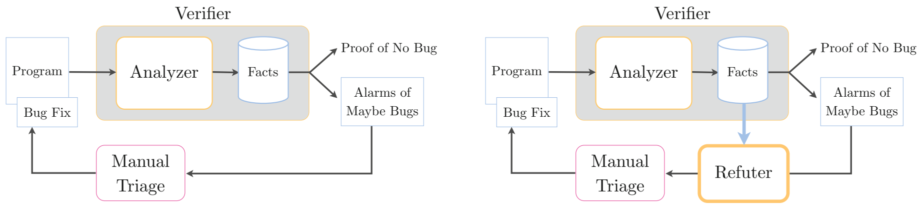


Fig. 1. From verification to alarm triage to bug fixes. The traditional setup (left) requires significant manual effort to triage alarms before bug fixes can be realized. Our approach (right) targets the alarm triage problem with an after-the-fact, automated refutation analysis that soundly filters out as many false alarms as possible to lower the manual triaging effort required. The automated algorithms are outlined in orange, the artifacts that they take as input or output are outlined in blue, and manual effort is outlined in red.

longer used by the operating system but cannot be freed by the garbage collector. For this client, we found the need for a highly precise heap reachability analysis capable of interprocedural path-sensitive reasoning with strong updates. This level of precision is quite difficult to achieve globally in an up-front manner, especially for programs with the size and complexity of our target applications (40K source lines of code plus 1.1M lines of the Android framework).

The conceptual architecture of our approach is diagrammed in Fig. 1. On the left, we show the traditional setup for verification emphasizing that if no proof is obtained, then the report of alarms must be triaged before bug fixes can be produced. On the right, we illustrate that our approach is to drop in a *refuter* stage between the alarm report and the manual triaging process. A refuter is an automated program analysis that tries to prove as many alarms false as possible in a *goal-directed* manner, thereby soundly filtering the list of alarms (hopefully significantly) that manual triaging must consider. For the Android leak client, the "Analyzer" in this architecture is instantiated with an off-the-shelf, state-of-the-art points-to analysis, and the "Refuter" is a tool called THRESHER that implements a highly precise symbolic analysis for refuting heap reachability queries.

In our terminology, a sound refuter is given a query assertion $\widehat{\sigma}$ (i.e., an abstraction of a set of concrete states) at a particular program point ℓ that the up-front analyzer is unable to verify. The goal of the refuter is in essence to derive a contradiction—to prove that there is no possible trace to program point ℓ with any concrete state satisfying $\widehat{\sigma}$. If successful, the alarm is false and can be filtered. Importantly, soundness of refutation analysis can be phrased in a partial correctness sense neither requiring witnessing a concrete trace to a state inside or outside the set abstracted by $\widehat{\sigma}$. In particular, not asking for witness traces enables coarser approximation while being sound with respect to refutations. In our setting, the abstract state $\widehat{\sigma}$ is a separation logic formula constraining a sub-heap, which permits the necessary strong update reasoning, and the analysis is a goal-directed, backwards symbolic analysis that over-approximates all paths to ℓ. The partial-correctness–style soundness criteria enables a relatively simplistic loop invariant inference procedure over separation constraints that turns out to be effective in practice for finding refutations.

While the refuter has the advantage of being goal-directed, the needed level of precision described above is still quite challenging to obtain while simultaneously achieving the required scalability. To achieve better scalability, one contribution of this work is a novel use of the up-front points-to analysis result (diagrammed in Fig. 1 as an arrow from "Facts" to "Refuter"). At a high-level, we enrich the abstract domain in the refutation analysis with a new pure constraint (a from constraint) that connects the heap abstraction during refutation analysis with the heap abstraction of the up-front points-to analysis. In essence, the from constraint enables reduction [4] with the points-to analysis result to derive contradictions earlier with less case splitting. This reduction is particularly important in our context, as case splitting is highly problematic for backwards analysis with separation constraints (see [3, Sect. 6]).

In the remainder of this extended abstract, we sketch (1) the soundness criteria for refutation analysis and (2) how the from constraint enables deriving contradictions earlier with less case splitting. Further details about our approach and methodology are described elsewhere [1, 2]. The bottom line result for our application to Android leak detection is that on a suite of seven Android applications ranging from 2K source lines of code to 40K lines plus the Android framework, THRESHER refuted 172 out of 196 false alarms while exposing 115 true bugs. In other words, THRESHER lowered the false alarm rate from 63% to 17% as compared to the points-to analysis alone—effectively filtering 88% of the false alarms reported by the points-to analysis.

Refutation Soundness

Concrete

$$\sigma \in \mathbf{State} \qquad s \in \mathbf{Statement} \qquad \langle \sigma, s \rangle \Downarrow \sigma'$$

Abstract

$$\widehat{\sigma} \in \widehat{\mathbf{State}} \qquad \gamma : \widehat{\mathbf{State}} \to \wp(\mathbf{State}) \qquad \vdash \{\widehat{\sigma}\}\ s\ \{\widehat{\sigma}'\}$$

We leave both the notion of concrete state σ and the programming language of interest mostly unspecified. The programming language is assumed only to be an imperative language of statements s defined by a big-step operational semantics judgment form $\langle \sigma, s \rangle \Downarrow \sigma'$ stating that in concrete state σ, evaluating statement s can result in a state σ'. The abstraction $\widehat{\sigma}$ is unspecified except that its meaning is given by a standard concretization function γ. And we write an unspecified abstract semantics for refuting queries as a judgment form deriving a Hoare triple $\vdash \{\widehat{\sigma}\}\ s\ \{\widehat{\sigma}'\}$ stating a pre-condition $\widehat{\sigma}$ and post-condition $\widehat{\sigma}'$ for a statement s.

The soundness condition for refutations in which we are interested can be stated as follows:

Condition 1 (Refutation Soundness)
If $\langle \sigma, s \rangle \Downarrow \sigma'$ *and* $\vdash \{\widehat{\sigma}\}\ s\ \{\widehat{\sigma}'\}$ *such that* $\sigma' \in \gamma(\widehat{\sigma}')$,
then $\sigma \in \gamma(\widehat{\sigma})$.

Informally, given a query post-condition $\widehat{\sigma}'$, we output a query pre-condition $\widehat{\sigma}$ such that if there is some execution of statement s to a concrete state σ' satisfying $\widehat{\sigma}'$, then that execution must begin in a concrete state σ satisfying $\widehat{\sigma}$. Note that the pre- and post-conditions are switched as compared to the standard statement of partial correctness.

This switching in the soundness condition is what we consider defining a goal-directed analysis.

If we derive $\bot$, the abstraction of the empty set of concrete states, for the query pre-condition $\widehat{\sigma}$, then there is no execution of statement s to a state satisfying the query post-condition $\widehat{\sigma}'$. In other words, we have derived a refutation. Our main point is not Condition 1 itself but that witnessing an execution is not required for deriving refutations.

Reducing Separation Constraints with Points-to Facts

The result of a (may) points-to analysis is a points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$. A vertex represents a set of possible memory addresses, typically those allocated at a particular static program point (under some context-sensitivity policy). Such a vertex is typically called an abstract location. An edge from vertex $\mathring{v}_1$ to $\mathring{v}_2$ states that there may be an execution from an address in the concretization of $\mathring{v}_1$ to an address in the concretization of $\mathring{v}_2$. Let us write $\mathring{r}$ for a set of vertices in a points-to graph (i.e., $\mathring{r} \subseteq \mathring{V}$ for a points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$).

In the refutation analysis, let us write $\widehat{a}$ for a symbolic address that abstracts a single concrete address. A from constraint

$$\widehat{a} \;\mathsf{from}\; \mathring{r}$$

relates a symbolic address and a set of abstract locations in the points-to graph in a rather expected way. The meaning is that the concrete address abstracted by $\widehat{a}$ must be in the set of concrete addresses abstracted by $\mathring{r}$ and can be axiomatized as follows for a given points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$:

$$\begin{aligned}
\widehat{a} \;\mathsf{from}\; \emptyset &\iff \mathsf{false} \\
\widehat{a} \;\mathsf{from}\; \mathring{V} &\iff \mathsf{true} \\
\widehat{a} \;\mathsf{from}\; \mathring{r}_1 \;\wedge\; \widehat{a} \;\mathsf{from}\; \mathring{r}_2 &\iff \widehat{a} \;\mathsf{from}\; \mathring{r}_1 \cap \mathring{r}_2 \\
\widehat{a} \;\mathsf{from}\; \mathring{r}_1 \;\vee\; \widehat{a} \;\mathsf{from}\; \mathring{r}_2 &\iff \widehat{a} \;\mathsf{from}\; \mathring{r}_1 \cup \mathring{r}_2
\end{aligned}$$

For our purposes, if we ever derive $\widehat{a}$ from $\emptyset$, then we have a derived refutation.

To see how from constraints enable earlier contradictions, consider the following example triple:

$$\vdash \left\{ \begin{array}{c} \big(\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p} \;\wedge\; \mathtt{x} \neq \mathtt{y}\big) \\ \vee \\ \big(\; \boxed{\mathtt{y} \;\mathsf{from}\; \mathsf{pt}_{\mathring{G}}(\mathtt{x}) \cap \mathsf{pt}_{\mathring{G}}(\mathtt{y})} \;\wedge\; \mathtt{x} = \mathtt{y}\big) \end{array} \right\} \quad \mathtt{x.f} := \mathtt{p} \quad \{\, \mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p} \,\}$$

Here, we write a program variable (e.g., `x`) for the value stored there. The query post-condition $\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p}$ states that we are considering a concrete state that has at least one memory location where the contents of field `y.f` is `p`, written $\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p}$. In the pre-condition, there are two cases to consider depending on whether or not `x` and `y` alias. In the first disjunct, we are looking for a way to reach the pre-location of the assignment $\mathtt{y}\cdot\mathtt{f} \mapsto \mathtt{p}$ with `x` and `y` not aliasing. In the second, we see if it is possible to reach the pre-location with x and y aliasing, which is a sufficient condition to imply the original query after the assignment. The shaded from constraint intersects the set of possible abstract locations of y with the points-to set of x, written $\mathrm{pt}_{\mathring{G}}(\mathtt{x})$. It says that this case is only possible if the intersection of the points-to set of x and the points-to set of y is non-empty. But even if this intersection is non-empty, the from constraint does not simply "check and forget" but retains whatever restriction obtained by considering the points-to set of x. Intuitively, the $\widehat{a}$ from $\mathring{r}$ constraint abstracts the set of storage locations through which the concrete address corresponding to $\widehat{a}$ flows.

Anecdotally, we have observed that from constraints are particularly important when branching from a method body to all of the call sites of the method (for context-sensitivity). Many of the disjuncts corresponding to the call sites are ruled out by a contradictory from constraint when intersecting with the points-to sets of the actual arguments. Intuitively, when such disjuncts are ruled out, we obtain a goal-directed form of object-sensitivity without the cost of analyzing the code between the argument's allocation site and the call site.

Acknowledgments. This work is joint with Sam Blackshear and Manu Sridharan. We thank the University of Colorado Programming Languages and Verification (CUPLV) group for many insightful discussions. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1055066.

References

1. Blackshear, S., Chang, B.-Y.E., Sankaranarayanan, S., Sridharan, M.: The flow-insensitive precision of Andersen's analysis in practice. In: Yahav, E. (ed.) SAS 2001. LNCS, vol. 6887, pp. 60–76. Springer, Heidelberg (2011)
2. Blackshear, S., Chang, B.Y.E., Sridharan, M.: Thresher: Precise refutations for heap reachability. In: Conference on Programming Language Design and Implementation (PLDI), pp. 275–286 (2013)
3. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6), 26 (2011)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Symposium on Principles of Programming Languages (POPL), pp. 269–282 (1979)

Cascade 2.0

Wei Wang, Clark Barrett, and Thomas Wies

New York University

Abstract. Cascade is a program static analysis tool developed at New York University. Cascade takes as input a program and a control file. The control file specifies one or more assertions to be checked together with restrictions on program behaviors. The tool generates verification conditions for the specified assertions and checks them using an SMT solver which either produces a proof or gives a concrete trace showing how an assertion can fail. Version 2.0 supports the majority of standard C features except for floating point. It can be used to verify both memory safety as well as user-defined assertions. In this paper, we describe the Cascade system including some of its distinguishing features such as its support for different memory models (trading off precision for scalability) and its ability to reason about linked data structures.

1 Introduction

Automatic verification using SMT solvers is an active area of research, with a number of tools emerging, such as ESC/Java [16], Caduceus [15], LLBMC [30], Spec# [1], HAVOC [8], VCC [10], LAV [29], and Frama-C [13]. Increasingly, SMT solvers are used as back-end checkers because of their speed, automation, and ability to model programs and assertions using built-in theory constructs.

Cascade[1] is an open-source tool developed at New York University for automatically reasoning about programs. An initial prototype of the system was described in [24]. This paper describes version 2.0, a from-scratch reimplementation which provides a number of new features, including support for nearly all of C (with the exception of floating point), support for loops and recursion via unrolling, support for loop invariants and deductive reasoning, and a new back-end interface supporting both CVC4 [2] and Z3 [14].[2] It is easy to add additional back-end plugins as long as they support the SMT-LIB input format.

In addition to describing the overall system, this paper focuses on two distinguishing features of Cascade: its support for multiple memory models and its extensibility for specific domains.

The paper is organized as follows. Section 2 gives an overview of the system and its features. Section 3 describes the three memory models supported by Cascade, and reports the results of an empirical evaluation of these models on

[1] Available at `http://cims.nyu.edu/~wwang1109/cascade/index.html`

[2] Another important feature of version 2.0 is that it has a very permissive license and if CVC4 is used, it does not depend on any code with restrictive licenses.

the NECLA suite of static analysis benchmarks [23]. Section 4 describes a case study in extending the system to reason about linked data structures, Section 5 describes related work, and Section 6 concludes.

2 System Design

Cascade is implemented in Java. The overall framework is illustrated in Figure 1. This version of Cascade focuses on C, but the system is designed to be able to accommodate multiple front-end languages. The C front-end converts a C program into an abstract syntax tree using a parser built using the Rats parser generator [17]. The core module takes an abstract syntax tree and a control file as input. The control file specifies one or more paths through the program, assumptions that should be made along the path, and assertions that should be checked along the path. The core module uses symbolic execution over the abstract syntax tree to build verification conditions corresponding to the assertions specified in the control file. Currently, it takes the approach of simple forward execution [3, 5, 18]. The core module converts paths through the abstract syntax tree into logical formulas.

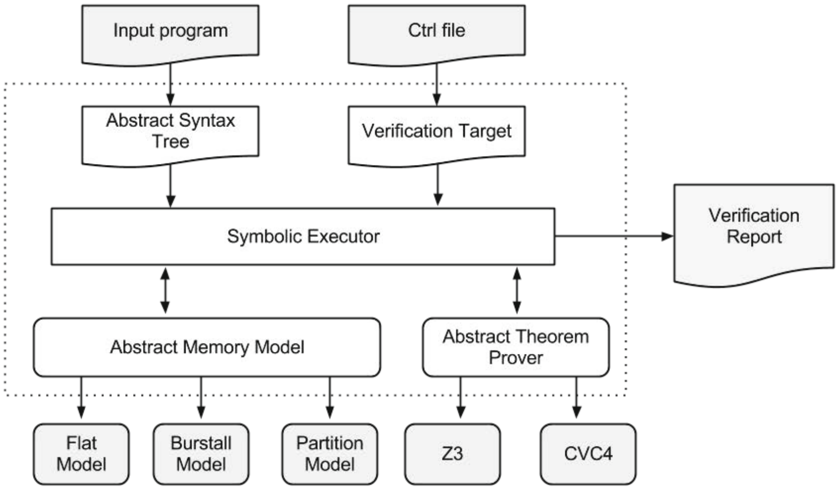


Fig. 1. Cascade framework

2.1 The Control File

Unlike some systems (e.g., [10], [8]), Cascade does not rely on annotated C code. Rather, a separate control file is used to guide the symbolic execution. Control files use XML and support the constructs detailed below. The rationale for the control file is that we want to be able to use Cascade on large existing code bases without having to modify the code itself.

Basic structure. Every control file begins with a sourceFile section that gives the paths to the source files. This is followed by one or more Run sections, each defining a constrained (symbolic) run of the program. Each run starts with a single startPosition command and ends with a single endPosition command that give respectively the start point and end point of the run. If the source code contains branches, Cascade will consider both branches by default (merging them when they meet again). If users wish to execute one branch in particular, they may include one or more wayPoint commands in run, to indicate the positions that the considered run should pass through. A simple example is shown in Figure 2.

```
int abs(int x) {
  int result;
  if(x>=0)
        result = x;
  else
        result = -x;
  return result;
}
```

```
<controlFile>
  <sourceFile name="abs.c" id="1" />
   <run>
    <startPosition fileId="1" line="1" />
    <wayPoint fileId="1" line="4" />
    <endPosition fileId="1" line="8" />
  </run>
  <run>
    <startPosition fileId="1" line="1" />
    <wayPoint fileId="1" line="6" />
    <endPosition fileId="1" line="8" />
  </run>
</controlFile>
```

Fig. 2. abs.c and abs.ctrl

Function Calls. Cascade supports procedure calls via inlining. Note that Cascade always assigns a unique name for each dynamically encountered variable declaration, so name clashes between caller and callee functions are not an issue. By default, Cascade can perform inlining and parameter passing automatically, as shown in Figure 3 (the body of function pow2 is inlined at the call sites in main). If users wish to specify a particular path in the function, a wayPoint command must be used to specify the line on which the function is called. Then, a function section can be embedded within the wayPoint command which provides an attribute funcName, as well as the wayPoints of the desired path inside the function. Even if multiple functions are called at the same line of code, this can be handled by nesting multiple function sections under the wayPoint command for that line of code. These function sections will constrain the function calls on that line in the order that the function calls appear (from left to right). Figure 4 gives an example.

Loops. By default, loops are eliminated using bounded loop unrolling [3]. A default number of unrolls can be specified on the command line, and a specific number of iterations for a particular loop can be specified using a loop command, as shown in Figure 5. Alternatively, a loop invariant can be specified using the invariant command as shown in Figure 6. If a loop invariant is provided, Cascade will simply check that the loop invariant holds when the loop is entered, and that it is preserved by a single iteration of the loop (for this second

```
int pow2(int x) {
  return x*x;
}

int main() {
  int a, b, result;
  a = 2;
  b = 3;
  result = pow2(a) + pow2(b);
  return result;
}
```

```
<controlFile>
  <sourceFile name="pow2.c" id="1" />
  <run>
    <startPosition fileId="1" line="5" />
    <endPosition fileId="1" line="11" />
  </run>
</controlFile>
```

Fig. 3. pow2.c and pow2.ctrl

```
int abs(int x) {
  int result;
  if(x>=0)
        result = x;
  else
        result = -x;
  return result;
}

int main() {
  int a, result;
  a = -4;
  result = abs(a) - abs(-a);
  return result;
}
```

```
<controlFile>
  <sourceFile name="absext2.c" id="1" />
  <run>
    <startPosition fileId="1" line="10" />
    <wayPoint fileId="1" line="13" >
      <function funcName="abs" funcId="1" >
        <wayPoint fileId="1" line="6" />
      </function>
      <function funcName="abs" funcId="2" >
        <wayPoint fileId="1" line="4" />
      </function>
    </wayPoint>
    <endPosition fileId="1" line="15" />
  </run>
</controlFile>
```

Fig. 4. absext2.c and absext2.ctrl

```
int log2(int num) {
  int result, i;
  result = 0;
  for(i=num; i>1; i=i/2) {
    result++;
  }
  return result;
}

int main() {
  int num, result;
  num = 1024;
  result = log2(num);
  return result;
}
```

```
<controlFile>
  <sourceFile name="log2.c" id="1" />
  <run>
    <startPosition fileId="1" line="10" />
    <wayPoint fileId="1" line="13" >
      <function funcName="log2" >
        <wayPoint fileId="1" line="4" >
          <loop iterTimes="10" />
        </wayPoint>
      </function>
    </wayPoint>
    <endPosition fileId="1" line="15" />
  </run>
</controlFile>
```

Fig. 5. log2.c and log2.ctrl

check, any variables updated in the loop body are assumed to be unconstrained so that the check is valid for all iterations of the loop). Then, the loop invariant is assumed going forward (this is in contrast to the default behavior which is to symbolically execute the loop a fixed number of times). As with assumptions and assertions (see below), invariants are specified using C expressions. Note that quantified loop invariants are acceptable, and Cascade's ability to solve them is limited only by the quantifier reasoning capabilities of the back-end solver.

```
int main() {
  int sum = 0;

  for (int i = 0; i<=10; i++) {
    sum = sum + i;
  }
  return sum;
}
```

```
<controlFile>
  <sourceFile name="forLoop_test.c" id="1" />
  <run>
    <startPosition fileId="1" line="1" />
    <wayPoint fileId="1" line="4" >
      <loop>
        <invariant><![CDATA[
        sum == (i-1) * i / 2 && i >= 0 && i <= 11
        ]]>
        </invariant>
      </loop>
    </wayPoint>
    <endPosition fileId="1" line="7" />
  </run>
</controlFile>
```

Fig. 6. sum.c and sum.ctrl

Commands. Two commands `cascade_assume` and `cascade_check` are provided, each of which takes a C expression as an argument. `cascade_assume` is used to constrain the set of possible states being considered to those satisfying the argument provided. `cascade_check` generates a verification condition to check that the symbolic execution up to this point satisfies the argument provided. Commands are allowed as part of a `startPosition`, `wayPoint`, or `endPosition` directive (see Figure 7).

```
int abs(int x) {
  int result;
  if(x>=0)
        result = x;
  else
        result = -x;
  return result;
}
```

```
<controlFile>
  <sourceFile name="absext.c" id="1" />
  <run>
    <startPosition fileId="1" line="1" />
    <endPosition fileId="1" line="7" >
      <command>
        <cascadeFunction> cascade_check
        </cascadeFunction>
        <argument><![CDATA[
        result >= 0
        ]]>
        </argument>
      </command>
    </endPosition>
  </run>
</controlFile>
```

Fig. 7. absext.c and absext.ctrl. The assertion is invalid due to the possibility of signed overflow.

As a new feature of Cascade 2.0, commands *can* also be included as annotations in the source code, as shown in Figure 8. To use this feature, the "`--inline-anno`" option must be enabled[3]. Cascade provides a number of extensions that can be embedded within C expressions to enable more expressive reasoning. Some are listed here:

– Logic symbols: $implies(P, Q)$, $forall(v, u, E)$ and $exists(v, u, E)$.
– Memory checks:
  - $valid(p)$: denotes that p is guaranteed to point to a memory address within a region allocated by the program.
  - $valid(p, size)$: denotes that the addresses from $p, ..., p + size - 1$ are valid (in the sense described above).
  - $valid\_malloc(p, size)$: denotes the assumptions that can be made on the pointer p after a `malloc` instruction.
  - $valid\_free(p)$: denotes that a `free` instruction on pointer p is admissible.

```c
int strlen(const char* str){
  ASSUME(valid_malloc(str,
                    4*sizeof(char)));

  int i=0;
  while(str[i] != '\0')
    ++i;

  ASSERT(forall(j,
         implies(j >= 0 && j <= i,
                 valid(&str[i]))));
  return i;
}
```

```xml
<controlFile>
  <sourceFile name="strlen.c"
              id="1" />
  <run>
    <startPosition fileId="1"
                   line="1" />
    <wayPoint fileId="1" line="7" >
      <loop iterTimes="3" />
    </wayPoint>
    <endPosition fileId="1"
                 line="13" />
  </run>
</controlFile>
```

Fig. 8. strlen.c and strlen.ctrl. Because we specify that the loop should be executed exactly 3 times, no errors are found.

3 Memory Models

One goal of Cascade is to support the analysis of systems software such as device drivers and operating systems code. These programs make heavy use of pointer manipulation and require a fairly precise memory model. A complementary goal of Cascade is to scale to large programs that are not as pointer-intensive. To achieve these complementary goals, Cascade provides three different memory models, with different trade-offs in terms of precision and scalability: (1) the *flat* model, in which all of memory is modeled as a single array; (2) the *Burstall* model [6] which uses an array for every different structure field; and (3) the *partition* model which divides up memory into several partitions, using a pointer analysis to ensure that variables that may alias end up in the same partition. In this section, we discuss these models in detail, including their semantics, implementation details, advantages and restrictions.

[3] The control file style annotation is designed to keep the source code clean, while the inline style is available for those users who prefer it.

3.1 Flat Memory Model

The flat memory model is essentially the standard conceptual memory model for C: the entire memory is represented as a single flat array M mapping addresses to values[4] (by default, both addresses and values are represented as fixed-width bit-vectors). Memory operations are modeled with the array operations *store* and *select*. Each program variable is modeled as the content of some address in memory. For example, the variable x is associated with a memory address $addr_x$, and all reads from and writes to x are done by accessing M at address $addr_x$. This model can soundly support all type-unsafe operations including union types, pointer arithmetic and pointer casts.

Concretely, we model the memory with two arrays M and *Size* of types

$$\begin{aligned} M &: \mathit{BitVec}(n) \rightarrow \mathit{BitVec}(m) \\ \mathit{Size} &: \mathit{BitVec}(n) \rightarrow \mathit{BitVec}(m) \end{aligned}$$

The constants m and n can be assigned on the command line via the options "`--mem-cell-size`" and "`--mem-addr-size`". [5]

To model dynamic memory allocation operations such as $x = \mathit{malloc}(\mathit{size})$, a fresh region variable region_x of type $\mathit{BitVec}(n)$ is created and stored at $addr_x$ of M. To keep track of the size of the allocated region, the auxiliary array variable *Size* is used to map region_x to *size*. Deallocation, $\mathit{free}(x)$, is modeled by selecting the region variable $M[addr_x]$ corresponding to x, and updating *Size* to 0 at $M[addr_x]$. In the initial state, the array *Size* is assumed to map all indices to 0. The following table gives the formal semantics of *malloc* and *free*.

Statement	Interpretation
$x = \mathit{malloc}(\mathit{size})$	$M' = \mathit{store}(M, addr_x, \mathit{region}_x)$ $\mathit{Size}' = \mathit{store}(\mathit{Size}, \mathit{region}_x, \mathit{size})$
$\mathit{free}(x)$	$\mathit{Size}' = \mathit{store}(\mathit{Size}, M[addr_x], 0)$

Note that the value of $\mathit{store}(a, i, v)$ is a new array equivalent to a except at index i where its value is now v [27]. The array M is the symbolic value of memory before the operation and M' is the symbolic value of memory afterwards. Having these definitions, memory checks can be formalized as follows:

$$\begin{aligned} &\mathit{valid}(p, \mathit{size}) \equiv \\ &\qquad \exists \mathit{region} : \mathit{BitVec}(n).\ \mathit{Size}[\mathit{region}] > 0 \implies \\ &\qquad\qquad M[addr_p] \geq \mathit{region} \ \wedge M[addr_p] + \mathit{size} \leq \mathit{region} + \mathit{Size}[\mathit{region}] \end{aligned}$$

The predicate $\mathit{valid}(p, \mathit{size})$ is inserted as an assertion before each memory access.

[4] Some tools use separate arrays for the stack and the heap. However, this is not always a sound assumption, so Cascade uses a single mapping to represent both.

[5] Integers are represented as fixed-size bit vectors, and thus integer arithmetic is arithmetic modulo 2^k where k is the number of bits. Cascade also allows the user to select unbounded integers to represent integers in the program. This is activated with the option "`--non-overflow`" .

Allocation and deallocation have associated guard predicates, *valid_malloc*, respectively, *valid_free*. The predicate *valid_free* is used to detect errors related to deallocation of invalid pointers. It is inserted as an assertion before each free instruction and is defined as follows:

$$valid\_free(x) \equiv M[addr_x] = 0 \vee Size[M[addr_x]] > 0$$

The predicate *valid_malloc* is used to ensure that the new region $region_x$ is indeed fresh and does not overlap with previously allocated regions. It is inserted as an assumption after each malloc instruction. Cascade provides two modes for the flat memory model that differ in how they interpret this predicate: an *unordered* and an *ordered* mode.

Unordered mode. The unordered mode can be selected with the command line option "`--sound`". In this mode, *valid_malloc* is interpreted as follows:

$$\begin{aligned}
&valid\_malloc(p, size) \equiv \\
&\quad M[addr_p] \neq 0 \implies M[addr_p] > 0 \wedge M[addr_p] \leq M[addr_p] + size \wedge \\
&\qquad (\forall region : BitVec(n).\ Size[region] > 0 \wedge region \neq M[addr_p] \implies \\
&\qquad\quad M[addr_p] + size \leq region \vee region + Size[region] \leq M[addr_p])
\end{aligned}$$

This interpretation accurately reflects the C semantics. However, the size of the allocation guards grow quadratically with the number of allocations encountered during symbolic execution (after instantiating the universal quantifiers with the actual regions). This places a high burden on the back-end SMT solvers.

Ordered mode. To obtain a more efficient SMT encoding, Cascade provides an additional ordered mode, which sacrifices precision for scalability without overly constraining the memory model. In this mode, Cascade assumes that every freshly allocated memory address is larger than the largest address in the latest allocated region. In order to track the latest allocated region, a new auxiliary variable *last_region* is introduced. This variable is updated appropriately after each allocation operation. The predicate *valid_malloc* is then interpreted as follows:

$$\begin{aligned}
&valid\_malloc(p, size) \equiv \\
&\quad M[addr_p] \neq 0 \implies M[addr_p] > 0 \wedge M[addr_p] \leq M[addr_p] + size \wedge \\
&\qquad (last\_region = 0 \vee last\_region + Size[last\_region] \leq M[addr_p])
\end{aligned}$$

Hence, in the ordered mode, the memory model does not capture memory management strategies in which freed addresses will be reallocated. However, this mode greatly reduces the size of the generated SMT solver queries without sacrificing much precision. In particular, many errors due to imprecise reasoning about pointer arithmetic between fields and objects can still be detected. Note that during symbolic execution, a data structure "Regions" is maintained to keep track of all allocated regions along the current path. Using this auxiliary data structure, we can completely instantiate the quantifiers in the guard predicates and memory checks.

3.2 Burstall Memory Model

The main idea of the Burstall memory model [6] is to split the memory according to the types of allocated objects, making the assumption that pointers with different types will never alias. Apart from common scalar types, each struct field is also represented as a unique type. This model guarantees that updates to different fields of a struct will not interfere with each other. Consequently, it cannot capture union types or pointer arithmetic on fields inside a struct object. Cascade has a preprocessor that detects such operations and gives a warning when using the Burstall model.

To encode the Burstall memory model in Cascade, M is encoded as a record instead of a flat array. Each record element represents the state of the memory for one type in the C program. The exact type of M is shown in Fig. 9. The number of record elements is bounded by the number of structure types defined in the C program. Note that for each record element, if its type is a pointer, the element type of the corresponding array is *Addr*; otherwise, it is *Scalar*.

$$\begin{array}{rlrl} \mathit{Ptr} : & \text{uninterpreted type} & \mathit{Scalar} : & \mathit{BitVec}(m) \\ \mathit{Offset} : & \mathit{BitVec}(n) & \mathit{Addr} : & \mathit{Ptr} \times \mathit{Offset} \end{array}$$

$$M : \mathit{Record} \left\{ \begin{array}{l} \mathit{type}_0 : (\mathit{Addr} \rightarrow \mathit{Addr} \mid \mathit{Scalar}), \\ \mathit{type}_1 : (\mathit{Addr} \rightarrow \mathit{Addr} \mid \mathit{Scalar}), \\ \ldots \\ \mathit{type}_k : (\mathit{Addr} \rightarrow \mathit{Addr} \mid \mathit{Scalar}) \end{array} \right\}$$

$$\mathit{Size} : \mathit{Record} \left\{ \begin{array}{l} \mathit{type}_0 : (\mathit{Addr} \rightarrow \mathit{Scalar}), \\ \mathit{type}_1 : (\mathit{Addr} \rightarrow \mathit{Scalar}), \\ \ldots \\ \mathit{type}_k : (\mathit{Addr} \rightarrow \mathit{Scalar}) \end{array} \right\}$$

Fig. 9. Types of auxiliary variables for the encoding of the Burstall memory model

3.3 Partition Memory Model

The partition memory model is a novel experimental model implemented in Cascade. We here provide only an abridged summary of this model since a detailed description is beyond the scope of this paper.

In the partition model, the memory is divided according to distinct program pointers. A valid pointer has ownership of the associated memory region. This model allows arbitrary pointer arithmetic inside a region, as well as dereferencing pointers to any location inside a region. The model further supports all untyped operations except pointer aliasing. For example, consider a program that non-deterministically assigns either `&x` or `&y` to a pointer variable `s`. A subsequent update of `x`, respectively, `y` via pointer `s` would not be detected if `x` and `y` are assigned to regions that are disjoint from the region of `s`. To obtain a memory partition that takes into account pointer aliasing, Cascade incorporates Steensgaard's unification-based pointer analysis [26] as a preprocessing step. Each set of potentially aliasing pointers is assigned to one region. For the above example, the preprocessor will assign `x` and `y` to the same region. In most cases, the number of pointer classes is much larger than the number of types in the C code. Hence, the partition model often provides a more fine-grained partition of the memory into disjoint regions compared to the Burstall model. This can significantly speed up the analysis in some cases, which we confirm in our experimental evaluation.

Similar to Burstall's model, the state of the memory is encoded as a record. The detailed types of the auxiliary variables are shown in Figure 10. Every region has its own array, and the element type of the array can be determined by the type of the pointers associated with that region.

$$M : Record \begin{Bmatrix} ptr_0 : BitVec(n) \rightarrow BitVec(m), \\ ptr_1 : BitVec(n) \rightarrow BitVec(m), \\ \ldots \\ ptr_k : BitVec(n) \rightarrow BitVec(m) \end{Bmatrix}$$

$$Size : Record \begin{Bmatrix} ptr_0 : BitVec(n) \rightarrow BitVec(m), \\ ptr_1 : BitVec(n) \rightarrow BitVec(m), \\ \ldots \\ ptr_k : BitVec(n) \rightarrow BitVec(m) \end{Bmatrix}$$

Fig. 10. Types of auxiliary variables for the encoding of the partition memory model

Initially, the record is empty. During symbolic execution, new record elements are added for new variable definitions. If the execution context changes its scope we can safely delete those elements associated with pointers not in the current scope. In this way, the memory state tracks only the active pointers in the current scope. This significantly simplifies the query formula given to the SMT solver. Note that both the unordered mode and the ordered mode used in the flat model can also be applied to each region in the partition memory model.

3.4 Evaluation

We report on a set of experiments using the multiple memory models in Cascade to check properties of the NECLA suite of static analysis benchmarks [23]. These benchmarks contain C programs demonstrating common programming situations that arise in practice such as interprocedural data-flow, aliasing, array allocation, array size propagation and so on. We excluded benchmarks relying on string library functions and floating point number calculations. The results of our experiments are summarized in Table 1 and Table 2.[6] Note that these benchmarks have also been used in other recent tool papers such as the one introducing LLBMC [30] (which also included evaluations of CBMC 3.8, CBMC

[6] More information on the experiments including the benchmarks and control files is available at `http://cims.nyu.edu/~wwang1109/cascade/vmcai.html`.

3.9 [9] and ESBMC 1.16 [12]) and another introducing LAV [29] (which also evaluated CBMC, ESBMC, and KLEE [7]). For comparison purposes, we report our results in a similar format to that shown in [29] and also show the best result reported there (in the LAV column).

The benchmark suite includes both faulty and correct programs. There are two notable discrepancies with the results reported by LAV. For benchmark ex10.c, LAV reports an error while Cascade does not. The reason is that we made an additional assumption, namely that a pointer passed into the main function was properly allocated. Without this assumption, Cascade would find the same invalid address access as did LAV. The other discrepancy was in benchmark ex40.c. In this program, a loop iterates over an array of size 100 until the value 0 is found. If the array does not have a 0 entry, an out-of-bounds violation will occur in the 101st iteration. Cascade finds this bug if enough loop iterations are examined.

While collecting the statistics, we have compared the performance of the different memory models in Cascade[7]. For the flat memory model, we did not observe a significant performance improvement of the ordered mode over the unordered mode in most of the benchmarks. This is because the size of the benchmarks is limited and so is the size of the queries given to the SMT solver. However, the results for some benchmarks with a large number of loop unrollings (ex17-100, ex26-200), or with invariant reasoning (ex1-inv, ex18-inv) are encouraging. We also found that the ordered mode is slightly slower than the unordered mode for benchmark ex23-36, but we have not yet investigated why this is so.

Furthermore, from the results, we can see that both the Burstall model and partition model scale much better than the flat model – they solved the benchmarks (ex7-200, ex18-100, ex21-100, and ex22-50) that timed out in either LAV or Cascade with the flat model (or both). And the overall performance of partition model is much better than Burstall. In particular, benchmark ex27-200 was solved with the partition model, but timed out with the Burstall model. Partition model is the default memory model in Cascade.

4 Reasoning about Linked Data Structures

In this section, we discuss how to extend Cascade to reason about properties of linked data structures. Analysis of such data structures typically requires a *reachability predicate* to capture the unbounded number of dynamically allocated cells present in a linked list. For a given address u, the reachability predicate characterizes the set of cells $\{u, u.f, u.f.f, \ldots\}$ reachable from u via continuously visiting field f.

4.1 Theory of Reachability in Linked Lists

LISBQ. Rakamarić et al. [21] presented a ground logic and an NP decision procedure for reasoning about reachability in liked list data structures. The logic

[7] Note that the programs in this benchmark suite are all type-safe – the Burstall model is accurate enough to detect all bugs.

Table 1. Evaluation on NECLA Benchmarks. The experiments were conducted on a 1.7GHz, 4GB machine running Mac OS. A timeout (indicated by *) of 600 seconds was set for each experiment. V indicates the program verification succeeded, and F indicates the program contains a bug which was detected by Cascade. In the third column, "inv" indicates that deductive reasoning with a loop invariant was used; a number indicates the number of loop unrollings used; "-" indicates either that the program is loop-free or that a failure occurs before any loops are entered; and "?" indicates the unknown default iteration times used by LAV. In the fourth and later columns, "-" indicates that there is no corresponding result for that loop configuration.

bnc.	F/V	#iter	Time(s)				
			LAV	Flat Model		Burstall Model	Partition Model
				Unordered	Ordered		
ex1	V	inv	-	68.561	50.079	0.645	**0.559**
		513	*	*	*	*	*
		3	0.35	0.895	0.663	0.512	0.771
ex2	V	inv	-	1.916	1.817	0.434	0.368
		1024	*	*	*	*	*
		3	0.47	0.534	0.535	0.332	0.42
ex3	F	inv	-	0.365	0.362	0.471	0.496
		10	-	0.576	0.552	0.558	0.774
		?	0.06	-	-	-	-
ex4	F	inv	-	0.39	0.425	0.375	0.452
		10	-	1.756	2.489	1.101	1.451
		?	0.24	-	-	-	-
ex5	V	-	0.02	0.136	0.132	0.114	0.109
ex6	V	-	0.11	0.187	0.134	0.159	0.142
ex7	V	inv	-	1.62	1.088	0.393	0.375
		200	*	*	*	9.266	**6.552**
		3	0.15	0.709	0.606	0.269	0.247
ex8	F	inv	-	0.173	0.128	0.193	0.193
		3	0.14	0.156	0.125	0.129	0.154
ex9	V	inv	-	0.666	0.768	0.452	0.441
		1024	*	*	*	*	*
		3	0.62	0.757	0.738	0.365	0.473
ex10	V	inv	-	2.795	2.605	0.94	1.115
		17	10.47	0.119	0.125	0.13	0.153
		3	1.14	0.763	1.407	0.471	0.689
ex11	V	3	0.08	0.215	0.222	0.211	0.215
ex12	F	10	-	1.099	0.985	0.724	0.948
		inv	-	0.363	0.376	0.428	0.381
		?	0.16	-	-	-	-
ex13	F	-	0.44	0.117	0.123	0.099	0.118
ex14	V	inv	-	0.344	0.332	0.338	0.304
		10	-	5.13	4.494	2.1	1.703
		?	0.13	-	-	-	-
ex15	V	-	0.34	1.731	1.522	0.235	0.233
ex16	F	inv	-	0.693	0.737	1.004	0.88
		4	-	0.927	0.929	0.955	0.931
		2	0.09F	0.22	0.232	0.266	1.82
ex17	V	inv	-	0.292	0.282	0.369	0.35
		100	-	19.274	17.068	29.193	**3.34**
		?	0.68	-	-	-	-

Table 2. Evaluation on NECLA Benchmarks (continued)

bnc.	F/V	#iter	Time(s)				
			LAV	Flat Model		Burstall Model	Partition Model
				Unordered	Ordered		
ex18	V	inv	-	424.414	65.531	**0.446**	0.804
		100	*	*	*	359.59	**7.066**
		10	3.0	8.088	9.412	2.527	1.298
ex19	F	inv	-	0.163	0.17	0.183	0.172
		3	0.08	0.371	0.424	0.466	0.387
ex20	F	inv	-	0.385	0.399	0.451	0.417
		1024	*	*	*	*	*
		1	0.32	0.498	0.389	0.315	0.29
ex21	V	inv	-	0.757	0.735	1.26	0.782
		100	-	*	*	**12.673**	16.2
		?	0.36	-	-	-	-
ex22	V	50	-	*	*	14.919	**12.133**
	V	?	4.1	-	-	-	-
ex23	V	inv	-	1.015	1.069	0.571	0.418
		36	6.46	27.44	34.0	**2.191**	2.847
ex25	F	inv	-	0.965	0.989	1.163	1.208
		3	0.20	1.654	1.046	0.876	1.506
ex26	F	inv	-	0.546	0.613	0.748	0.687
		200	-	31.122	28.062	20.314	**7.877**
		?	0.62	-	-	-	-
ex27	F	inv	-	1.584	1.913	1.219	4.68
		200	-	*	*	*	**55.585**
		?	5.28	-	-	-	-
ex30	F	-	-	0.134	0.147	0.596	0.395
		?	0.24	-	-	-	-
ex31	V	inv	-	0.308	0.312	0.348	0.317
		7	5.62	0.62	0.976	0.624	0.432
ex32	V	inv	-	0.892	0.826	1.067	0.608
		1000	-	*	*	*	*
		?	0.5	-	-	-	-
ex34	V	-	0.24	0.416	3.141	0.441	0.508
ex37	F	-	0.20	0.107	0.143	0.131	0.132
ex39	F	inv	-	0.228	0.238	0.292	0.255
		3	0.07	0.306	0.273	0.641	0.307
ex40	F	inv	-	0.323	0.288	0.336	0.314
	V	3	0.10	0.345	0.307	0.271	0.25
ex41	F	inv	-	0.24	0.23	0.267	0.25
		3	0.44	0.515	0.332	0.214	0.25
ex43	F	-	-	1.793	1.318	1.041	1.271
		inv	-	0.907	0.857	0.731	0.794
		?	17.91	-	-	-	-
ex46	F	3	*	0.127	0.124	0.174	0.159
ex47	F	inv	-	12.925	9.365	4.563	1.779
		2	1.38	0.315	0.449	0.192	0.235
ex49	F	inv	-	0.33	0.339	0.414	0.378
	V	3	0.08	0.24	0.246	0.235	0.204
inf1	F	-	0.22	0.261	0.29	0.261	0.282
inf2	F	-	1.25	0.223	0.158	0.185	0.195
inf4	F	-	0.38	0.403	0.502	0.345	0.662
inf5	F	-	0.15	0.221	0.235	0.15	0.209
inf6	V	-	0.12	0.247	0.243	0.207	0.216
inf8	V	-	0.19	0.333	0.417	0.323	0.439

provides a ternary predicate $x \xrightarrow{f} z \xrightarrow{f} y$, which we refer to as the *between predicate*. The predicate states that cell y is reachable from cell x via field f, yet, not without going through cell z first. In other words, z is between x and y. Binary reachability $\mathit{reach}(f, x, y)$ via field f can then be expressed as $x \xrightarrow{f} y \xrightarrow{f} y$. The between predicate enables precise tracking of reachability information during symbolic execution of heap updates that modify field f (potentially creating cycles in the heap). In [19], Lahiri and Qadeer presented the logic of interpreted sets and bounded quantification (LISBQ), which includes the between predicate but also admits reasoning about the content of unbounded list data structures. They showed that LISBQ is still decidable in NP using a decision procedure that builds on an SMT solver.

LISBQ as a local theory extension. More recently, we explored the connection between Lahiri and Qadeer's result to local theory extensions [25]. A theory extension is a first-order theory that is defined by extending a base theory with additional symbols and axioms. For example, the theory of arrays over integer indices can be formalized as a theory extension where the base theory is the theory of linear integer arithmetic, the extension symbols are the array store and select functions, and the extension axioms are McCarthy's select over store axioms.

A theory $\mathcal{T}$ is called *local* if satisfiability modulo $\mathcal{T}$ can be decided by reduction to the base theory via local instantiation of the extension axioms. Here, local instantiation means that only those axiom instances are considered that do not introduce new terms to the input formula. Local theory extensions are interesting because they provide completeness guarantees for the quantifier instantiation heuristics implemented in modern SMT solvers, and at the same time give a simple syntactic restriction on the kinds of axiom instances that need to be considered.

In [28], we showed that LISBQ can be formalized as a local theory extension. This yields an interesting generalization of previous results in [19, 21]. For example, the base theory can now provide an interpretation of memory cells, e.g., as bitvectors, which results in a theory of reachability that admits address arithmetic. Another generalization obtained this way is to interpret fields as arrays. The formulas generated during symbolic execution in [19, 21] can grow exponentially in the number of store operations $x.f := y$ along the executed path. The encoding of fields as arrays avoids this exponential blowup by deferring case splits on store operations to the SMT solver. Finally, the connection to local theory extensions also provides new possibilities to further improve the efficiency of SMT-based decision procedures for LISBQ.

4.2 Linked Lists in Cascade

We have explored some of these possibilities in the context of Cascade. We added a reachability predicate to the C expression language usable in assumptions and assertions. The axioms of LISBQ are encoded via a new theory axiom encoding

```c
#define NULL (int *) 0

typedef struct NodeStruct {
  struct NodeStruct *next;
  int data;
} Node;

void append(Node *l1, Node *l2) {
  ASSUME(create_acyclic_list(l1, 5)
      && create_acyclic_list(l2, 5));

  Node *l = l1;
  Node *e = l1;
  Node *last = NULL;

  while (e) {
    last = e;
    e = e->next;
  }

  if (!last)
    l = l2;
  else
    last->next = l2;

  ASSERT(reach(next, l1, l2));
}
```

```xml
<controlFile>
  <sourceFile name="list_append.c"
              id="1" />
  <run>
    <startPosition fileId="1"
                   line="8" />
    <wayPoint fileId="1" line="16" >
      <loop iterTimes="5" />
    </wayPoint>
    <wayPoint fileId="1" line="21" />
    <endPosition fileId="1"
                 line="27" />
  </run>
</controlFile>
```

Fig. 11. list_append.c and list_append.ctrl. The predicate *create_acyclic_list*$(l, 5)$ indicates that l is a singly-linked list of size 5. The predicate *reach*$(next, l1, l2)$ indicates that $l1$ can reach $l2$ by following the link field *next*.

module. The base theory interprets memory cells as bitvectors of fixed width in order to model pointer arithmetic; and fields are interpreted as arrays mapping bitvector indices to bitvector values. Cascade then instantiates the LISBQ theory axioms for the ground terms appearing in each SMT query and hands the axiom instances together with the query to the SMT solver.

In order to keep the input formula to the SMT solver relatively small, we exploit our results on locality: we only partially instantiate the theory axioms. That is, we only instantiate quantified variables that appear below function symbols in the axioms, while keeping the remaining variables quantified. The resulting quantified formulas fall into fragments for which the quantifier instantiation heuristics that are implemented in SMT solvers are guaranteed to be decision procedures. Partial instantiation provides a good compromise between an approach where we only rely on the solver's heuristics and do not instantiate axioms upfront, and an approach where we fully instantiate the axioms and do not use the heuristics in the solver at all. The former is typically fast but incomplete on satisfiable input formulas, the latter is complete but typically slow. This is confirmed by our experimental evaluation. In fact, often partial instantiation yields the best running time.

Example. Figure 11 shows the code and control file of a small list-manipulating procedure in C that appends two lists together. In this test case, we are interested in verifying that after the procedure returns, the head of the first list `l1` can reach the head of the second list `l2`.

Evaluation. In order to evaluate Cascade's new ability to reason about linked data structures, we chose two suites of benchmark programs manipulating singly-linked lists (SL) and doubly-linked lists (DL), respectively. In these benchmarks, various nontrivial reachability-related assertions are checked. The evaluation was performed on a 1.7GHz, 4GB machine running Mac OS. For each benchmark, the time limit was set to 60 seconds. The results appear in Table 3. We used three different instantiation heuristics for the quantified axioms, and partial instantiation is much more efficient than the other two options in most cases. This is noteworthy considering that SMT solvers use sophisticated instantiation heuristics internally.

Note that for the DL benchmarks with invalid assertions, both no instantiation and full instantiation time out most of the time, while partial instantiation reports "unknown" immediately. For benchmarks with quantifiers that don't fall into a know complete fragment, an "unknown" indicates that the SMT solver was unable to find a proof using its instantiation heuristics. Thus, an unknown result should be considered the same as an invalid (satisfiable) result with the understanding that it could be a false positive. In other words, an immediate "unknown" result is the best we could hope for in this situation.

Table 3. Results on singly- and doubly-linked list benchmarks. A timeout (indicated by *) of 600 seconds was set for each experiment. NI is for no instantiation, PI is for partial instantiation, and FI is for full instantiation. The superscript UN indicates that the result from the SMT solver is "unknown".

Benchmark	F/V	NI	PI	FI	Benchmark	F/V	NI	PI	FI
sl_append	V	0.02	**0.02**	0.08	dl_append	V	**0.08**	0.34	0.14
sl_contains	V	**0.01**	0.01	0.05	dl_contains	V	**0.01**	0.03	0.02
sl_create	V	0.13	**0.05**	3.12	dl_create	V	12.98	**0.18**	1.81
sl_filter	F	0.11	**0.08**	0.13	dl_filter	F	*	0.31^{UN}	*
sl_findPrev	V	**0.02**	0.05	0.09	dl_findPrev	V	**0.05**	0.07	0.06
sl_getLast	V	0.01	**0.01**	0.06	dl_getLast	V	3.34	**0.02**	1.27
sl_insertBefore	V	**0.20**	0.26	3.03	dl_insertBefore	V	*	**2.74**	*
sl_partition	F	0.08	**0.08**	0.09	dl_partition	F	*	0.23^{UN}	*
sl_remove	F	1.34	**0.07**	1.76	dl_remove	F	*	0.68^{UN}	*
sl_removeLast	F	0.03	**0.02**	0.06	dl_removeLast	F	*	0.11^{UN}	*
sl_reverse	V	0.09	**0.06**	1.57	dl_reverse	V	*	**18.65**	*
sl_split	F	0.05	**0.03**	0.07	dl_split	F	*	0.11^{UN}	*
sl_traverse	V	0.03	**0.01**	0.52	dl_traverse	V	8.68	**0.02**	*

5 Related Work

In the last decade, a variety of SAT/SMT-based automatic verifiers for C programs have been developed, such as bounded model checkers (CBMC [9], ESBMC [12], LLBMC [30], LAV [29], Corral [20] and Cascade), symbolic execution tools (KLEE [7]), and modular verifiers (VCC [10], HAVOC [8], and Frama-C [13]). In most cases, these tools use either flat memory models (e.g., CBMC, LLBMC, ESBMC, LLBMC, LAV, KLEE and early versions of VCC), or Burstall-style memory models (e.g., Corral and Caduceus [15]).

As we have discussed, these models force the user to choose between scalability and being able to capture the effects of type-unsafe behaviors. The VCC developers proposed a typed memory model that attempts to strike a balance between scalability and precision [11]. This model maintains a set of valid pointers with disjoint memory locations, and restricts memory accesses only to them. Special code annotation commands called split and join are introduced to switch between a typed mode and a flat mode. However, the additional axioms introduced for the mode switching slow down reasoning [4]. Böhme et al. use a variant of the VCC model [4] but few details are given. A variant of Burstall's model is proposed in [22]. It employs a type unification strategy that simply removes the uniqueness of relative type constants when detecting type casts. However, this optimization is too coarse to handle code with even mild use of low-level address manipulations and type casts, as the memory model will quickly degrade into the flat model.

Frama-C also develops several memory models at various abstraction levels: Hoare, typed, and flat models. As an optimization strategy, Frama-C mixes the Hoare model and flat model by categorizing variables into two classes: logical variables and pointer variables. The Hoare model is used to handle the logical variables and the flat model manages the pointer variables. This strategy is similar to our partition model. However, our partition model provides a more fine-grained partition for the pointer variables.

Our partition model is similar to the memory model of VCC that divides memory based on various pointers. The main difference is that we map the pointers to separately updatable memory regions, and thus ease the burden of SMT axiomatization for distinguishing pointers. Steensgaard's pointer analysis is incorporated to control the issue of pointer aliasing. Compared to the VCC model, our modeling seems more natural – we can detect untyped operations before memory splitting, and thus avoid switching between typed and flat modes. The direct performance comparison is difficult because of VCC's contract based approach to verification. However, results from [4] seem to confirm the folk wisdom that splitting the heap into disjoint regions performs best.

6 Conclusion

In this paper, we presented the latest version of Cascade, an automatic verifier for C programs. It supports multiple memory models in order to balance efficiency and precision in various ways. Our empirical evaluation shows that Cascade is competitive with other tools. Furthermore, we have shown that with a modest effort, it can be extended to reason about simple properties of linked data structures.

In the future, we will integrate an invariant inference engine to relieve the annotation burden on users. Moreover, we are planning to support procedure contracts that enable local reasoning via frame rules.

Acknowledgments. We would like to thank Christopher Conway, Morgan Deters, Dejan Jovanović, and Tim King for their contributions to the design and implementation of the Cascade tool. This work was supported by NSF grants CCF-0644299 and CCS-1320583.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
3. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of Design Automation Conference (DAC 1999), vol. 317, pp. 226–320 (1999)
4. Böhme, S., Moskal, M.: Heaps and data structures: A challenge for automated provers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 177–191. Springer, Heidelberg (2011)
5. Brand, D., Joyner, W.H.: Verification of protocols using symbolic execution. Comput. Networks 2, 351 (1978)
6. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, 23–50 (1972)
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI 2008, pp. 209–224 (2008)
8. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
10. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
11. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for c. ENTCS 254, 85–103 (2009)
12. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ansi-c software. In: ASE, pp. 137–148 (2009)
13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c a software analysis perspective (2012)
14. 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)
15. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)

16. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
17. Grimm, R.: Rats!, a parser generator supporting extensible syntax (2009)
18. King, J.C.: Symbolic execution and program testing. Communications of the ACM 385, 226–394 (1976)
19. Lahiri, S.K., Qadeer, S.: Back to the future. Revisting precise program verification using SMT solvers. In: POPL, pp. 171–182 (2008)
20. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
21. Rakamarić, Z., Bingham, J.D., Hu, A.J.: An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 106–121. Springer, Heidelberg (2007)
22. Rakamarić, Z., Hu, A.J.: A scalable memory model for low-level code. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 290–304. Springer, Heidelberg (2009)
23. Sankaranarayanan, S.: Necla static analysis benchmarks (2009)
24. Sethi, N., Barrett, C.W.: Cascade: C assertion checker and deductive engine. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 166–169. Springer, Heidelberg (2006)
25. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. Logical Methods in Computer Science 4, 4 (2008)
26. Steensgaard, B.: Points-to analysis in almost linear time. In: ACM Symposium on Principles of Programming Languages, pp. 32–41 (1996)
27. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A decision procedure for an extensional theory of arrays. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, p. 29 (2001)
28. Totla, N., Wies, T.: Complete instantiation-based interpolation. In: POPL (2013)
29. Vujošević-Janičić, M., Kuncak, V.: Development and evaluation of LAV: An SMT-based error finding platform. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 98–113. Springer, Heidelberg (2012)
30. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT Softw. Eng., 29 (2004)

A Logic-Based Framework for Verifying Consensus Algorithms[⋆]

Cezara Drăgoi[1], Thomas A. Henzinger[1], Helmut Veith[2], Josef Widder[2], and Damien Zufferey[3,⋆⋆]

[1] IST Austria
[2] TU Wien, Austria
[3] MIT CSAIL

Abstract. Fault-tolerant distributed algorithms play an important role in ensuring the reliability of many software applications. In this paper we consider distributed algorithms whose computations are organized in rounds. To verify the correctness of such algorithms, we reason about (i) properties (such as invariants) of the state, (ii) the transitions controlled by the algorithm, and (iii) the communication graph. We introduce a logic that addresses these points, and contains set comprehensions with cardinality constraints, function symbols to describe the local states of each process, and a limited form of quantifier alternation to express the verification conditions. We show its use in automating the verification of consensus algorithms. In particular, we give a semi-decision procedure for the unsatisfiability problem of the logic and identify a decidable fragment.We successfully applied our framework to verify the correctness of a variety of consensus algorithms tolerant to both benign faults (message loss, process crashes) and value faults (message corruption).

1 Introduction

Fault-tolerant distributed algorithms play a critical role in many applications ranging from embedded systems [12] to data center management [8,14]. The development of these algorithms has not benefited from the recent progress in automated reasoning and the vast majority of the correctness proofs of these algorithms is still written by hand. A central problem that these algorithms solve is the consensus problem in which distributed agents have initial values and must eventually decide on some value. Moreover, processes must agree on a common value from the set of initial values, even in environments that contain faults and uncertainty in the timing of events. Charron-Bost and Schiper [10] introduced the *heard-of* model as a common framework to model different assumptions on the environment, and to express the most relevant consensus algorithms from the literature. We introduce a new logic $\mathbb{CL}$ tailored for the heard-of model.

[⋆] Supported by the National Research Network RiSE of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant PROSEED and by the ERC Advanced Grant QUAREM.

[⋆⋆] Damien Zufferey was at IST Austria when this work was done.

The heard-of model is a round-based computational model: conceptually, processes operate in lock-step, and distributed algorithms consist of rules that determine the new state of a process depending on the state at the beginning of the round and the messages received by the process in the current round. The work in [10] introduces the notion of *heard-of* set $HO(p, r)$, which contains the processes from which some process p may receive messages in a given round r. Without restricting the heard-of sets, it could be the case that they are all empty, i.e., that there is no communication, and it is obvious that no interesting distributed computing problem can be solved. In [10] a way to describe meaningful communication is introduced, namely via communication predicates that constrain the heard-of sets in the computation. For instance, in a system consisting of n processes, the communication predicate $\forall r \forall p.\, |HO(p, r)| > n/2$ states that in all rounds all processes can receive messages from a majority of processes. As is the case in this example, the quantification over rounds is typically used in a way that corresponds to a fragment of linear temporal logic, using only simple combinations of the "globally" and "finally" operators. We can thus eliminate the round numbers and rewrite the above example as $\Box(\forall p.\, |HO(p)| > n/2)$, and call terms like $\forall p.\, |HO(p)| > n/2$ *topology predicates*, as they restrict the communication graph in a round. It is demonstrated in [10] that many consensus algorithms from the literature can be expressed in this framework. These algorithms are correct only for specific communication predicates.

Our goal is to automate Hoare-style reasoning for distributed algorithms in the heard-of model. To this end, we have to define a logic that has a semi-decision procedure for satisfiability, and is able to capture properties of the states and the effect of the transitions. For instance, our logic must be able to capture topology predicates such as

"each process receives messages from at least $n - t$ processes,"

where n and t are integer variables that model the parameters of the system, such as the number of processes and faulty processes. Moreover, the logic should describe the values of the variables manipulated by the processes. For example

"if a process p decides on a value v, then a majority of processes currently have v stored in their `x` variable."

Finally, we have to capture the transitions of the algorithms, for instance

"all processes that receive a value v from more than two thirds of the processes, set variable `x` to v."

We thus need a logic that allows universal quantification over processes, defining sets of processes depending on the values of their variables, and linear constraints on the cardinalities of such sets of processes. These constraints can be expressed in first order logic, but since the satisfiability problem is undecidable, we need to find a logic that strikes a balance between expressiveness and decidability.

Contributions. We introduce a multi-sorted first-order logic called *Consensus verification logic* $\mathbb{CL}$ whose formulas express topology predicates and constrain the values of the processes' local variables using: (1) set comprehensions, (2) cardinality constraints, (3) uninterpreted functions, and (4) universal quantification. To automate the check of verification conditions we introduce a semi-decision procedure for unsatisfiability. This procedure soundly reduces checking the validity of implications between formulas in $\mathbb{CL}$ to checking the satisfiability of a set of formulas in Presburger arithmetics and a set of quantifier-free formulas with uninterpreted function symbols. The latter two have a decidable satisfiability problem. Furthermore, we have identified a fragment of the logic for which the satisfiability problem is decidable. The proof is based on a small model argument. We have successfully applied the semi-decision procedure to a number of consensus algorithms from the literature. In particular, we have applied it to all algorithms from [10], which surveys the most relevant (partially synchronous) consensus algorithms in the presence of benign faults, including a variant of Paxos. In addition we applied it to the algorithms from [4], which tolerate value faults, and to a basic synchronous consensus algorithm from [19].

2 Fault-Tolerant Distributed Algorithms in the HO-Model

In this section, we present the class of distributed algorithms we want to verify. These are algorithms in the *heard-of* model of distributed computations [10]. In the following, we introduce an adaptation of the heard-of model, suitable for automated verification. Distributed algorithms consist of n processes which interact by message passing, where n is a parameter. The executions are organized in rounds, and we model each round to consist of two transitions.

In the first transition, called *environment transition*, processes communicate by exchanging messages and intuitively an adversary, called *environment*, determines for each process the set of processes it receives messages from, i.e., its heard-of set. In a variant of the heard of model [10], the environment also assigns to each process a coordinator process. In the second transition, called *computation transition*, processes change their local state depending of the messages received in the previous phase. These transitions update disjoint sets of variables: the variables updated by the environment, in the first transition of a round, are called *environment variables*, the variables updated by the processes, in the second transition, are called *computation variables*. In the following we describe the variables of the distributed algorithm and the semantics of the two types of transitions.

Variables: The local variables manipulated by the distributed algorithm are of type `process`, `set of processes` or of data types, e.g., `integer` or `boolean`. The variables of type process and sets of processes are the environment variables, denoted *EVars*. The heard-of set of a process is represented by a local variable of type set of processes. Similarly, the coordinator of a process is represented by

```
Init(int $v_p$){ x := $v_p$; dec := ?; }

Comp :
  S: send x to all processes
  U: if received more than $2n/3$
     messages,
       then x := the smallest most often
                        received value;

       if more than $2n/3$ received values
            are equal with x
       then dec := x
```

(a) Algorithm.

EVars ::=	*HO* of type set of processes
CVars ::=	`x`, `dec` of type integer

(b) Process local variables.

TP_s ::=	"true"
TP_t^1 ::=	"there is a set A with more than $2n/3$ processes s.t. all processes receive the messages sent by the processes in A"
TP_t^2 ::=	"all processes receive the messages sent by more than $2n/3$ processes"

(c) Topology predicates; n represents the size of the network

"All the executions where:
 (1) $\exists$ an environment transition satisfying TP_t^1, and after that
 (2) $\exists$ an environment transition satisfying TP_t^2
solve Consensus."

(d) Specification.

Fig. 1. A round based algorithm in the HO-model that solves Consensus

a local variable of type process. The variables of data types are called computation variables, denoted *CVars*. For some distributed algorithms, we use global variables, *GVars*, of integer type to model round numbers. For simplicity of presentation, although of data type, we consider the global variables as environment variables that are deterministically incremented in the environment transitions.

Environment Transitions: The environment transitions assigns non-deterministically values to the environment variables of each process.

Computation Transition: Computation transitions assign values to the local computation variables of processes. These assignments are guarded by `if-then-else` statements. The latter contain conditions over the local state of the process and the messages received. In our view of the heard-of model we regard messages as values of the local variables of data type of other processes. The set of messages received by a process is determined by the value of its environment variables (*HO*-sets) and the `send` statements executed by the other processes. These statements are of the form "`send` *var* `to` *destination*", e.g. "`send x to all processes`" or "`send x to coordinator`'; they are parametrized by the variables sent, `x`, and the destination processes. More precisely, a process p receives `x` from process q, if q is in the heard-of set of p, and q executes "`send x`" and p is a destination process of this send statement.

Executions: A state of the distributed algorithm is defined by an n-tuple of local process states, and a valuation for the global variables, if there are any. The local state of a process is defined by a valuation of its variables. A computation starts with an initialization round, *Init*, followed by a sequence of rounds, *Comp*. The executions of a typical distributed algorithm are sequences of the form

$[p_1.\mathit{Init}(v_1)||\dots||p_n.\mathit{Init}(v_n)];\ \big(\mathit{Env};[p_1.\mathit{Comp}||\dots||p_n.\mathit{Comp}];\big)^*$ where Env is an environment transition, *Init* and *Round* are defined in Fig. 1, n is the number of processes, $||$ is the parallel composition, $p.R$ states that process p is executing R, '$*$' is the Kleene iteration of the sequential composition, and v_i, for $1 \leq i \leq n$, are integers different from a distinguished integer denoted by '?'.

Example 1. The distributed algorithm in Fig. 1 consists of n processes, each of them having two local variables `x` and `dec` of integer type, and one environment variable, the HO-set. The *computation* transitions are given in Fig. 1a. For each process, the *Init* transition initializes `dec` to a special value '?' and `x` to an input value. In the other rounds, all processes execute *Comp*. Given a process p, the values of the `x` variables of each process q in $HO(p)$ defines a multiset. It corresponds to the messages received by p.

The first `if` statement means that if p receives messages from more than two thirds of the processes, it updates its local variable `x` to the minimal most often received value. If the condition does not hold, the value of `x` stays unchanged. As the HO-set at different processes may differ, it can be that only some processes update `x`. In the second `if` statement, a process p updates the value of the variable `dec` if it received the same value from more then two thirds of the processes. As two thirds of the processes have the same value, there is a majority around this value.

3 Verification of Distributed Algorithms

Specifying Consensus. Intuitively, a distributed algorithm solves consensus if starting from an initial state where each process p has a value, it reaches a state where all the processes agree on one of the initial values. More precisely, consensus is the conjunction of four properties: *agreement*, no two process decide differently, *validity*, if all processes start with v then v is the only possible decision, *irrevocability*, any decision is irrevocable, and *termination*, eventually all processes decide. It is well-known from literature [22] that consensus cannot be solved if the environment transitions are not restricted. Hence, the specifications we consider are actually conditional. In the literature, the conditions are given in natural language and we express them with *topology predicates* and temporal logic formulas over these predicates. More precisely, topology predicates are conditions on the environment variables. We use topology predicates to restrict the effect of an environment transition, i.e., they restrict the domain of the non-deterministic assignments. To restrict the environment transitions in an execution, we use very simple LTL formulas: we consider conjunctions, where the first conjunct has the form $\Box\phi$, and the second conjunct is of the form $\Diamond(\phi_1 \wedge \Diamond(\phi_2 \wedge \Diamond(\ldots \wedge \Diamond(\phi_\ell))))$, where ϕ, ϕ_1, $\ldots \phi_\ell$ are topology predicates.

Example 2. The system in Fig. 1 solves Consensus by making all processes agree on the valuation of `dec`. Its specification is given in Fig. 1d. It uses three topology predicates, TP_s, TP_t^1 and TP_t^2, given in Fig. 1c. In temporal logic parlance, agreement can be stated as $\Box\, \mathit{Agrm}$, where Agrm says that "for any two processes p, q, either one of them has not decided, i.e., $\texttt{dec} = ?$ or they decide the same value, i.e., $\texttt{dec}(p) = \texttt{dec}(q) \neq ?$" (1)

Termination can be stated as $\Diamond\, Term$, where $Term$ says that "for all processes p, $\texttt{dec}(p) \neq ?$". To ensure termination, the distributed algorithm in Fig. 1 requires the existence of two specific rounds satisfying the topology predates TP_t^1 and TP_t^2. The specification is then given by $\Box\, TP_s \Rightarrow \Box\, Agrm$ and

$$\Big(\Box\, TP_s \wedge \big(\Diamond\, (TP_t^1 \wedge \Diamond TP_t^2)\big)\Big) \Rightarrow \Diamond\, Term.$$

Invariant Checking for Distributed Algorithms. We consider a logic-based framework to verify that a distributed algorithm satisfies its specification, where formulas represent sets of states or binary relations between states.

To prove the safety properties, i.e., agreement, validity, and irrevocability[1], we use the *invariant checking* approach, i.e., given a formula Inv_s that describes a set of states of the system, we check that Inv_s is an *inductive invariant* for the set of computations where all states satisfy the topology predicate TP_s and that Inv_s implies the three safety properties of consensus. The proof that Inv_s is an inductive invariant reduces to checking that the initial states of the system satisfy Inv_s and checking that the following holds:

$$\big(Inv_s(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{a}) \wedge TP_s(\boldsymbol{p}, \boldsymbol{e}) \wedge TR(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{a}, \boldsymbol{a'})\big) \Rightarrow Inv_s(\boldsymbol{p}, \boldsymbol{e'}, \boldsymbol{a'})$$

where $\boldsymbol{p}$ is the vector of processes, $\boldsymbol{e}$ is the vector of environment variables, $\boldsymbol{a}$ is the vector of computation variables, $TP_s(\boldsymbol{p}, \boldsymbol{e})$ is a topology predicate, and $TR(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{a}, \boldsymbol{a'})$ is the transition relation associated with an environment transition or a computation transition (unprimed and primed variables represent the value of the variables before and after a transition, respectively).

In our example, the invariant Inv_s states that

"no process has decided or there is a value v such that a majority of processes store the value v in their local variable x *and all processes that have decided chose v as their decision value".* (2)

To prove termination, our technique targets specifications that require a bounded number of constrained environment transitions. W.l.o.g. let r_1 and r_2 be the special rounds required for termination such that r_1 happens before r_2. For simplicity of presentation, we assume that both rounds satisfy the same topology predicate TP_t. To prove termination, the user must provide an inductive invariant, denoted Inv_t, that holds between the two special rounds, that is:

$$\big(Inv_s(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{a}) \wedge TP_t(\boldsymbol{p}, \boldsymbol{e}) \wedge TP_s(\boldsymbol{p}, \boldsymbol{e}) \wedge TR(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{a}, \boldsymbol{a'})\big) \Rightarrow Inv_t(\boldsymbol{p}, \boldsymbol{e'}, \boldsymbol{a'})$$

$$\big(Inv_t(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{a}) \wedge TP_s(\boldsymbol{p}, \boldsymbol{e}) \wedge TR(\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{a}, \boldsymbol{a'})\big) \Rightarrow Inv_t(\boldsymbol{p}, \boldsymbol{e'}, \boldsymbol{a'})$$

[1] Irrevocability can be stated as a property of the transition relation. It requires the use of a relational semantics for the round computations.

Moreover, this invariant has to be strong enough to achieve termination when the second special round happens, that is:

$$\big(Inv_t(\boldsymbol{p},\boldsymbol{e},\boldsymbol{a}) \wedge TP_t(\boldsymbol{p},\boldsymbol{e}) \wedge TP_s(\boldsymbol{p},\boldsymbol{e}) \wedge TR(\boldsymbol{p},\boldsymbol{e},\boldsymbol{e'},\boldsymbol{a},\boldsymbol{a'})\big) \Rightarrow Term(\boldsymbol{p},\boldsymbol{e'},\boldsymbol{a'}).$$

In our running example, the invariant for termination Inv_t is a stronger version of the safety invariant, and states that "there exists a value v such that the local variable `x` of any process equals v".

4 Consensus Verification Logic $\mathbb{CL}$

In this section, we introduce our logic $\mathbb{CL}$ that formalizes topology predicates, state properties, and the transition relation. We first introduce a graph-based representation for the states of the distributed algorithms we consider. Then, we define the syntax and semantics of our logic, whose formulas are interpreted over the graph-based representation.

4.1 Graph-Based Representation of States

We model states by *network graphs*, where each node represents a process. Node and link labels correspond to the values of the computation variables and environment variables, respectively. Formally, network graphs are tuples $G = (N, E, L_N, L_E)$, where N is a finite set of nodes, $L_N : N \times CVars \to \mathcal{D}$ defines a labeling of nodes with values from a potentially unbounded domain $\mathcal{D}$, E is a set of edges, and $L_E : E \rightharpoonup 2^{EVars}$ defines a labeling of the edges. For any environment variable `ev` $\in EVars$ of `process` type, the edges labeled by `ev` define a *total* function over the nodes in the graph (i.e., each node has exactly one successor defined by an edge labeled by `ev`). The heard-of sets are represented by variables of type `set of processes`; they do not define a total function because a node can have multiple or no successors w.r.t. the label HO.

A *state* of a distributed algorithm is a pair $C = (G, \nu)$, where G is a network graph and $\nu : GVars \to \mathcal{D}$ is a valuation of the global variables. Relations between two network states of the same system are represented by pairs (G, ν), where the vocabulary of the labels is doubled by introducing their primed versions. As we are interested in relations between states that belong to the same execution, the two states contain exactly the same set of processes.

Fig. 2a shows a state with three processes of the algorithm in Fig. 1, and Fig. 2b shows a relation between two states of the same algorithm. For simplicity, we draw only the labeled edges and omit the `dec` variable.

4.2 Syntax and Semantics

We define a multi-sorted first-order logic, called *Consensus verification logic* $\mathbb{CL}$, to express properties of sets of states (e.g., invariants) or relations between states (transition relations). The syntax of the logic is given in Fig. 3. The logic has

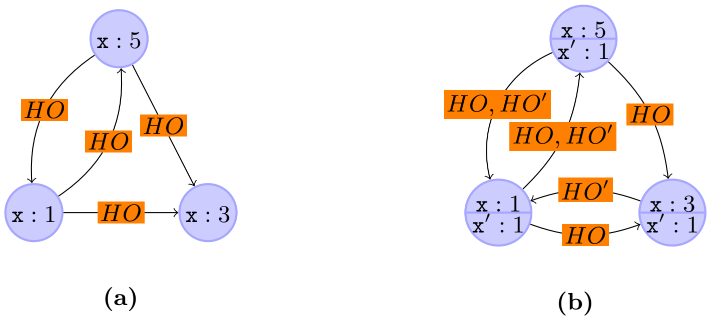


Fig. 2. Sample state (a) and relation between states (b)

	Sort P	Sort 2^P	Sort D	Sort $\mathbb{Z}$
Function symbols	$f : P \to P$	$F : P \to 2^P$	$\mathtt{x} : P \to D$	$\lvert \cdot \rvert : 2^P \to \mathbb{Z}$
Variables	$p,\ q$	$S,\ A$	$v,\ \Theta$	$N,\ n$
Terms	$t_P ::= p,\ q,\ f(p)$	$t_S ::= S, F(p), t_S \cap t_S$	$t_D ::= v,\ \mathtt{x}(p)$	$t_{\mathbb{Z}} ::= \lvert t_S \rvert,\ N,\ n$
Atomic formulas	$\varphi_P ::= t_P^1 = t_P^2$ $t_P \in t_S$,	$\varphi_S ::= t_S^1 \subseteq t_S^2$	$\varphi_D ::= t_D^1 \leq t_D^2$	$\varphi_{\mathbb{Z}} ::=$ linear constraint over $t_{\mathbb{Z}}$

Set comprehensions $\{q \mid \forall \boldsymbol{t}.\, \varphi(q, \boldsymbol{t}, \boldsymbol{p})\}$, where $\varphi ::= \varphi_P \mid \varphi_D \mid \varphi_{\mathbb{Z}} \mid \neg\varphi \mid \varphi \wedge \varphi$

Universally quantified formulas $\psi^{\forall}(\boldsymbol{S}) :: = \forall \boldsymbol{p}.\, \mathbb{B}(\{p \in S \mid p \in \boldsymbol{p}, S \in \boldsymbol{S}\}) \Rightarrow \mathbb{B}^+(\varphi_D)$, where

$\boldsymbol{p}$ is a set of process variables, $\boldsymbol{S}$ is a set of set variables and given a set of formulas Γ, $\mathbb{B}(\Gamma)$, resp. $\mathbb{B}^+(\Gamma)$, is a boolean combination, resp., positive boolean combination, of formulas in Γ

$\psi ::= \varphi_P \mid \varphi_S \mid \varphi_{\mathbb{Z}} \mid \varphi_D \mid \psi^{\forall} \mid \psi \wedge \psi \mid \neg\psi,$ $\psi_{\mathbb{CL}} ::= \psi \mid \exists p.\, \psi_{\mathbb{CL}} \mid \exists v.\, \psi_{\mathbb{CL}}$

where set comprehensions can be used as set terms

Fig. 3. Syntax of $\mathbb{CL}$ formulas, defined by $\psi_{\mathbb{CL}}$

four sorts: process, denoted P, sets of processes, denoted 2^P, integers, denoted $\mathbb{Z}$, and data, denoted D. We write $F[\varphi(*)](p)$ instead of $F(p) \cap \{q \mid \varphi(q)\}$.

The models of a formula in $\mathbb{CL}$ are pairs (G, μ), where $G = (N, E, L_N, L_E)$ is a network graph and μ is a valuation of the free variables. In the following, we describe the semantics of $\mathbb{CL}$ formulas and their use. We use the convention that the global variables correspond to free variables of formulas. The satisfaction relation is denoted by $G \models_\mu \varphi$. The interpretation of a term t w.r.t. (G, μ) is denoted by $[\![t]\!]_{(G,\mu)}$.

Atomic Formulas over Terms of Sort Process: The terms of sort P are built using a set of function symbols Σ_{pr} of type $P \to P$. They are interpreted as nodes in the graph, e.g., for any variable $p \in P$, $[\![p]\!]_{(G,\mu)}$ is a node in the graph

G. The interpretation of the function symbols is defined by the labeled edges, i.e., $[\![f(p)]\!]_{(G,\mu)} = u$ iff the graph G contains an edge $([\![p]\!]_{(G,\mu)}, u)$ labeled by f. The only predicate over terms of type P is equality.

We use the function symbols in Σ_{pr} for two purposes. First, they represent the values of local environment variables of type process, such as the coordinator of a process. Second, we use them to model processes in the heard-of sets with distinguished local states, such as the processes storing the minimal value, or the value with the most occurrences in the considered set.

Atomic Formulas over Terms of Sort Data: The terms of sort D are interpreted as values of the data domain $\mathcal{D}$. The node labels in G, i.e., the values of the computation variables, are represented in the logic by a set of function symbols $\mathtt{x} : P \to D$, one for each node label/computation variable. That is, $[\![\mathtt{x}(p)]\!]_{(G,\mu)} = d$ iff $d \in \mathcal{D}$ is the label $\mathtt{x}$ of the node $[\![p]\!]_{(G,\mu)}$, i.e., $L_N([\![p]\!]_{(G,\mu)}, \mathtt{x}) = d$. We assume that the domain $\mathcal{D}$ is totally ordered. The predicates over data terms are non-strict comparison and equality.

Atomic Formulas over Terms of Sort Set: The terms of sort 2^P are interpreted as sets of processes, i.e., sets of nodes in the graph. They are built using a set of function symbols Σ_{set} of type $P \to 2^P$. For any function symbol $F : P \to 2^P$ in Σ_{set}, $[\![F(p)]\!]_{(G,\mu)}$ is a set of nodes from N such that $u \in [\![F(p)]\!]_{(G,\mu)}$ iff F is one of the labels of the edge $(u, [\![p]\!]_{(G,\mu)}) \in E$. The heard-of sets are modeled using a function symbol $HO \in \Sigma_{set}$, where $[\![HO(p)]\!]_{(G,\mu)}$ is the set of nodes representing the processes $[\![p]\!]_{(G,\mu)}$ hears from. The logic contains the inclusion predicate over set terms and the membership predicate over process and set terms.

Atomic Formulas over Terms of Sort Integer: The atomic formulas over $\mathbb{Z}$-terms are linear inequalities and they constrain the cardinality of the set terms. We consider a distinguished integer variable n, which is interpreted as the number of processes in the network. For example, $|HO(p)| > 2n/3$ states that the process p receives messages sent by more than two thirds of the processes, and $\big|HO\big[\mathtt{x}(*) = \mathtt{x}(p)\big](p)\big| > 2n/3$ states that the value $\mathtt{x}(p)$ is received more than $2n/3$ times by process p.

One of the key features of the logic are the *set comprehensions*. They are used in the invariants to state that a majority is formed around one value, in the topology predicates to identify the set of processes that every one hears from, and in the transition relation to identify the processes that will update their local state. A set comprehension is defined by $\{q \mid \rho(q)\}$, where ρ is a (universally quantified) formula that contains at least one occurrence of the variable q (representing processes in the set). For ease of notation, we associate with each set comprehension a unique set variable used in a formula as a macro for its definition. The interpretation of a set comprehension is $[\![\{q \mid \rho(q)\}]\!]_{(G,\mu)} = \{u \in N \mid G \models_{\mu[q \leftarrow u]} \rho(q)\}$.

Set Comprehensions with Quantifier-Free Formulas: Typically, invariants identify sets of processes whose local variables have the same value. For example, the invariant Inv_s in (2) is defined using the set of processes whose local variable x equals v, i.e., $S_V = \{q \mid \mathtt{x}(q) = v\}$. Topology predicates are also expressed using set comprehension: the two topology predicates from Fig. 1c are expressed in $\mathbb{CL}$ by:

$$\begin{aligned} TP_t^1 &::= |A| > 2n/3 \;\wedge\; |S_A| = n, \text{ with } S_A = \{q \mid HO(q) = A\} \\ TP_t^2 &::= |S_{HO}| = n, \text{ with } S_{HO} = \{q \mid |HO(q)| > 2n/3\}, \end{aligned}$$

where A is a set variable, S_A is the set of processes whose heard-of set equals A, and S_{HO} is the set of processes that receive from more than $2n/3$ processes.

Set Comprehensions with Universally Quantified Formulas: Typical examples of such set comprehensions used in topology predicates are: the kernel $K = \{q \mid \forall t.\, q \in HO(t)\}$, i.e., the set of processes every one hears from and $S_{no\_split} = \{q \mid \forall t.\, |HO(t) \cap HO(q)| \geq 1\}$, which is the set of processes that share some received message with any other process in the network.

Set comprehensions are also used to select the processes that update their local state. Typically, the value assigned to some local variable is chosen from the received ones, e.g., the minimal received value, or the minimal most often received value. To express such updates, the process in the HO-set that holds such a value is represented as the value of a function symbol in Σ_{pr}. For example, the first update from the algorithm in Fig. 1a can be written as $\mathtt{x}'(p) = \mathtt{x}(mMoR(p))$, where $mMoR(p)$ is interpreted as a process q s.t. $\mathtt{x}(q)$ is the minimal most often received value by p. This constraint over the interpretation of $mMoR$ can be expressed by $|S| = n$ (we assume that all processes have sent the value of their x variable), where

$$S = \left\{ \begin{array}{l} q \mid \forall t.\, t \in HO(q) \Rightarrow \\ \qquad \left(\begin{array}{r} |HO[\mathtt{x}(*) = \mathtt{x}(mMoR(q))](q)| = |HO[\mathtt{x}(*) = \mathtt{x}(t)](q)| \Rightarrow \\ \mathtt{x}(mMoR(q)) \leq \mathtt{x}(t) \\ \wedge |HO[\mathtt{x}(*) = \mathtt{x}(mMoR(q))](q)| \geq |HO[\mathtt{x}(*) = \mathtt{x}(t)](q)| \end{array} \right) \end{array} \right\} \tag{3}$$

Above, S represents the set of processes q s.t. $\mathtt{x}(mMoR(q))$ is interpreted as the minimal most often received value by q. If $|S| = n$, i.e., S contains all the processes in the network, then for all processes q, the x variable of $mMoR(q)$ equals the minimal most often received value by q.

Universally Quantified Formulas: The universally quantified formulas in $\mathbb{CL}$ are implications, where (1) quantification is applied only over process variables, (2) the left hand side of the implication is a boolean combination of membership constraints, and (3) the right hand side of the implication is a positive boolean combination (without negation) of atomic formulas over data. For example, the transition relation for the algorithm in Fig. 1 is expressed by

$$\begin{aligned} TR \;=\; &\forall p.\, p \in S_{HO} \Rightarrow \mathtt{x}'(p) = mMoR(p) \;\wedge\; \forall p.\, p \in S_{HV} \Rightarrow \mathtt{dec}'(p) = \mathtt{x}'(p) \;\wedge \\ &\forall p.\, p \notin S_{HO} \Rightarrow \mathtt{x}'(p) = \mathtt{x}(p) \;\wedge\; \forall p.\, p \notin S_{HV} \Rightarrow \mathtt{dec}'(p) = \mathtt{dec}(p), \end{aligned} \tag{4}$$

where $S_{HV} = \{q \mid |HO[\mathtt{x}(*) = \mathtt{x}'(q)](q)| > 2n/3\}$ and S_{HO} is defined above.

The state properties in the definition of consensus, e.g., *Agrm* given by (1) in Sec. 3, are expressed using universally quantified formulas:

$$Agrm \ = \ |S| = n \wedge \forall p, q.\, p, q \in S \Rightarrow \mathtt{dec}(p) = \mathtt{dec}(q). \tag{5}$$

Remark 1. The formulas that express the guarded assignments, the inductive invariants, and the properties that define consensus, are in the form of universally-quantified implications. The left-hand side of these implications is typically more involved. To express these formulas, $\mathbb{CL}$ restricts the syntax of universally-quantified implications in a way that is sufficient to express the formulas we encountered. Note that these constraints on the use of universal variables can be overpassed using set comprehensions, e.g., $\forall t, q.\, \mathtt{x}(t) \neq \mathtt{x}(q)$ is equivalent to $S = \{q \mid \forall t.\, \mathtt{x}(t) \neq \mathtt{x}(q)\} \wedge |S| = n$.

Finally, $\mathbb{CL}$ formulas are existentially-quantified boolean combinations of atomic formulas and universally quantified formulas.

To conclude let us formalize the definition of the invariants, Inv_s and Inv_t given in Sec. 3, required to prove the correctness of the system in Fig. 1.

$$\begin{aligned}
Inv_s &= Inv_s^1 \vee \exists v.\, Inv_s^2(v), \text{where} \\
&\qquad Inv_s^1 = \forall q.\, \mathtt{dec}[q] = ? \text{ and} \\
&\qquad Inv_s^2(v) = |S_V| > 2n/3 \wedge \forall q.\, \mathtt{dec}(q) = ? \vee \mathtt{dec}(q) = v = \mathtt{x}(q) \\
Inv_t &= \exists v\, \forall q.\, \mathtt{x}(p) = v \wedge \big(\mathtt{dec}(q) = ? \vee \mathtt{dec}(q) = v = \mathtt{x}(q)\big)
\end{aligned} \tag{6}$$

Verification condition for distributed algorithms are implications between $\mathbb{CL}$ formulas, such as the ones in Sect. 3, where the invariants, transition relations, and properties are expressed in $\mathbb{CL}$.

5 A Semi-decision Procedure for Implications

Classically, checking the validity of a formula is reduced to checking the unsatisfiability of its negation. Since $\mathbb{CL}$ is not closed under negation, the negation of an implication between $\mathbb{CL}$ formulas is not necessarily in $\mathbb{CL}$. In this section, we will present (1) a sound reduction from the validity of an entailment between two formulas in $\mathbb{CL}$ to the unsatisfiability of a formula in $\mathbb{CL}$, (2) a semi-decision procedure for the unsatisfiability problem in $\mathbb{CL}$, and (3) identify a fragment $\mathbb{CL}_{dec}$ of $\mathbb{CL}$ which is decidable.

5.1 Reducing Entailment Checking in $\mathbb{CL}$ to Unsatisfiability

Let us consider the following entailment $\varphi \Rightarrow \psi$ between $\mathbb{CL}$ formulas φ and ψ. There are two reasons why $\varphi \wedge \neg\psi$ might not belong to $\mathbb{CL}$. First, if ψ has a sub-formula of the form $\exists^*\forall^*$, then by negation, the quantifier alternation becomes $\forall^*\exists^*$, which is not allowed in $\mathbb{CL}$. Second, the restricted use of universally quantified variables in $\mathbb{CL}$ is not preserved by negating the constraints on the existential variables of ψ, e.g., if $\psi = \exists p_1, p_2. p_1 = p_2$, then its negation is not in $\mathbb{CL}$ because difference constraints between universally quantified variables are not allowed. We define a procedure, which receives as input an implication $\varphi \Rightarrow \psi$ between two formulas in $\mathbb{CL}$. The algorithm we define hereafter builds a new formula ϕ from $\varphi \wedge \neg\psi$. It restricts the interpretation of the universally quantified variables that do not satisfy the syntactical requirements of $\mathbb{CL}$ to terms built over the existentially quantified variables.

Reduction Procedure: The formula ϕ is built in three steps. In the first step, the formula $\varphi \Rightarrow \psi$ is transformed into an equivalent formula $\phi_1 = \exists\xi.\, (\varphi_1 \wedge \psi_1)$ where all the existential quantifiers appear at the beginning (φ_1 and ψ_1 are equivalent to φ and $\neg\psi$, respectively, modulo some renaming of existentially-quantified variables; also, ψ_1 is transformed such that no universal quantifier appears under the scope of a negation). The second step consists of identifying the set of universally quantified variables $\boldsymbol{\beta}$ in ψ_1 that appear in sub-formulas not obeying the syntactic restrictions of $\mathbb{CL}$.

In the last step, let $\mathcal{T}(\xi)$ be the set of terms over the variables in ξ that contain at most k occurrences of the function symbols from $\mathbb{CL}$, where k is the maximum number of function symbols from a term of the formula $\varphi \wedge \neg\psi$. Then, ϕ is obtained by restricting the interpretation of the universally quantified variables in $\boldsymbol{\beta}$ to the domain defined by the interpretation of the terms in $\mathcal{T}(\xi)$:

$$\phi = \exists\xi. \bigwedge_{\gamma \in [\boldsymbol{\beta} \to \mathcal{T}(\xi)]} \left(\varphi_1 \wedge \psi_1 \left[\begin{array}{c} \beta \leftarrow \gamma(\beta) \\ \text{for every } \beta \in \boldsymbol{\beta} \end{array} \right] \right) \tag{7}$$

Lemma 1. *Let $\varphi \Rightarrow \psi$ be an implication between two formulas in $\mathbb{CL}$, and ϕ the formula in (7). The unsatisfiability of ϕ implies the validity of $\varphi \Rightarrow \psi$.*

Note that if ψ has no existentially quantified variables, then the unsatisfiability of ϕ is equivalent to the validity of $\varphi \Rightarrow \psi$.

Rationale: All the state properties used to define consensus, e.g., *Agrm* in (5) from Sec. 4, are expressible using universally-quantified formulas in $\mathbb{CL}$. Thus, for checking that an invariant implies these properties, the reduction procedure is sound and complete. This is not necessarily true for the verification conditions needed to prove the inductiveness of an invariant, that is, verification conditions of the form $\varphi \Rightarrow \psi$, where ψ is an inductive invariant. In the following, we give evidences for the precision of the reduction procedure in these cases.

In systems that solve consensus, all the computations contain a transition after which only one decision value is possible [11]. Therefore, the set of reachable states can be partitioned into two: the states where any value held by a process may become the decision, and the states where there is a unique value v that can be decided; often this corresponds to a majority formed around v. This implies that the inductive invariants are usually a disjunction of two formulas, one for each set of states described above. In the negation of the invariant, the universally quantified variables that do not obey to the restrictions in $\mathbb{CL}$ are those used to express that there is no value on which all processes agree.

In all our examples, the two sets of states are demarcated by the existence of at least one process that has decided.[2] Given invariants in this disjunctive form, to prove them inductive w.r.t. a transition TR, two situations have to be considered: (1) no process has decided before applying TR and at least one process has decided after TR, and (2) some processes decided before applying TR. In (1), to prove the unsatisfiability of $\varphi \wedge \neg\psi$ it is sufficient to map the universally quantified variables in $\neg\psi$ on terms denoting the value of one of the processes that have decided, and in (2), it is sufficient to map them on the terms denoting the values around which a majority was formed before applying TR.

Example 3. To prove that Inv_s, given in (6), is an invariant w.r.t. the transition relation TR given in (4), more precisely the case where no process decided before applying TR, one needs to prove the validity of $(Inv_s^1 \wedge TR) \Rightarrow Inv_s[\mathtt{dec} \leftarrow \mathtt{dec}']$, where $Inv_s[\mathtt{dec} \leftarrow \mathtt{dec}']$ is the obtained from Inv_s by substituting the function symbol dec with the function symbol dec′. This is equivalent with proving the unsatisfiability of the following formula, where we have expanded the definition of $Inv_s[\mathtt{dec} \leftarrow \mathtt{dec}']$:

$$Inv_s^1 \;\wedge\; TR \;\wedge\; \exists p.\, \mathtt{dec}'(p) \neq ? \;\wedge\; \forall v.\, \underbrace{\neg Inv_s^2(v)[\mathtt{dec} \leftarrow \mathtt{dec}']}_{\rho(v)}. \tag{8}$$

Notice that $\forall v.\, \neg Inv_s^2(v)[\mathtt{dec} \leftarrow \mathtt{dec}']$ does not belong to $\mathbb{CL}$ because it contains a $\forall\exists$ quantifier alternation. In this case, we soundly reduce the unsatisfiability of (8) to the unsatisfiability of

$$Inv_s^1 \;\wedge\; TR \;\wedge\; \exists p.\, \mathtt{dec}'(p) \neq ? \wedge \rho(v)[v \leftarrow \mathtt{dec}'(p)]. \tag{9}$$

by restricting the interpretation of universally quantified variable v to the value decided by process p, i.e., $\mathtt{dec}'(p)$.

If some processes decided, the term denoting the value around which a majority was formed before applying TR is the existentially quantified v in Inv_s.

5.2 Semi-decision Procedure for Unsatisfiability

In this section, we present the semi-decision procedure for the unsatisfiability problem in $\mathbb{CL}$. This procedure soundly reduces the unsatisfiability of a $\mathbb{CL}$ formula to the unsatisfiability of a quantifier-free Presburger formula (cardinality constraints) or the unsatisfiability of a formula with uninterpreted functions and order constraints (constraints on data). The satisfiability of these formulas is decidable and checkable using an SMT solver.

We give an overview on the main steps, Step 1 to Step 5, of the semi-decision procedure on an example, before we formalize them.

Overview: Let us consider the formula in (9), stating that no process decided before applying the transition relation TR given in (4), and afterwards two processes decide on different values:

[2] The only exception is the *LastVoting* algorithm, where the demarcation includes also the existence of a process having a local variable (not the decision one) set to true.

$$\varphi \;=\; \forall t.\, \mathtt{dec}(t) = ? \;\wedge\; TR \;\wedge\; \mathtt{dec}'(p) \neq ? \wedge \mathtt{dec}'(q) \neq ? \wedge \mathtt{dec}'(p) \neq \mathtt{dec}'(q)$$

The semi-decision procedure starts by instantiating universal quantifiers and set comprehension over the free variables of φ. This strengthens the data and cardinality constraints over terms with free variables (see Step 3).

In our example, the cardinality constraints are strengthened by instantiating the universal quantification in TR and the definition of the set comprehension S_{HV}, over the free variables p and q. The processes denoted by p and q decide in the round described by TR, therefore these variables belong to the set S_{HV}; from the definition of S_{HV}, the value decided by each of them, i.e., $\mathtt{x}'(p)$, resp., $\mathtt{x}'(q)$, was received from at least two thirds of the processes in the network, i.e., $|HO[\mathtt{x}(*) = \mathtt{x}'(p)](p)| > 2n/3$ and $|HO[\mathtt{x}(*) = \mathtt{x}'(q)](q)| > 2n/3$. The semi-decision procedure builds a Presburger formula from the cardinality constrains that use set terms over p and q; the definitions of the sets are abstracted. The obtained formula is $kp > 2n/3 \wedge kq > 2n/3$, where $kp = |HO[\mathtt{x}(*) = \mathtt{x}'(p)](p)|$ and $kq = |HO[\mathtt{x}(*) = \mathtt{x}'(q)](q)|$ (see Step 4). This formula is a satisfiable.

Then, the semi-decision procedure checks the satisfiability of the quantifier-free formula with uninterpreted function symbols defined by the data constraints over terms with free variables (Step 5). In our example this formula is $\mathtt{dec}'(p) = \mathtt{x}'(p) \;\wedge\; \mathtt{dec}'(q) = \mathtt{x}'(q) \;\wedge\; \mathtt{dec}'(p) \neq \mathtt{dec}'(q)$, and is also satisfiable.

Therefore, for $\mathbb{CL}$ formulas, restricting the interpretation of universal quantifiers to free variables is not sufficient to derive contradictions. The reason is that cardinality constraints induce relations between set comprehentions, which are neither captured by the Presburger formula nor the quantifier-free data formula. Notice that due to cardinality constraints, which state that each of the sets $HO[\mathtt{x}(*) = \mathtt{x}'(p)](p)$, resp. $HO[\mathtt{x}(*) = \mathtt{x}'(q)](q)$, contains more then two thirds of the processes in the network, their intersection is non-empty. Therefore there exists a process, r, which belongs to both sets. Instantiating the definitions of these sets over r reveals that $\mathtt{x}(r) = \mathtt{x}'(p)$ and $\mathtt{x}(r) = \mathtt{x}'(q)$, which contradicts the hypothesis that p and q decided on different values.

Thus, if the Presburger formula is satisfiable, it is used to discover relations between set comprehension. This formula is used to check which intersections or differences of set variables are non-empty; for each non-empty region, φ is extended with a free variable representing a process of this region (see Step 4). The semi-decision procedure is restarted using the new formula constructed from φ.

Semi-decision Procedure: Let $\varphi = \forall \boldsymbol{y}.\, \psi$ be a $\mathbb{CL}$ formula in prenex normal form, where $\boldsymbol{y}$ is a tuple of process variables. W.l.o.g., we assume that the formula does not contain existential quantifiers, only free variables. Formally, the procedure to check the unsatisfiability of φ iterates over the following sequence of steps:

Step 1: introduce fresh process variables for the application of function symbols over free variables. Let $\varphi_1 = \forall \boldsymbol{y}.\, \psi_1$, be the formula obtained after this step.

Step 2: enumerate truth valuations for set membership over free variables and instantiate set comprehensions. Let $\varphi_2 = \bigvee \forall \boldsymbol{y_2} \psi_2$, where each disjoint corresponds to a truth valuation. Notice that new quantified formulas are introduced in this step. Let $S = \{q \mid \rho(q)\}$ be a set comprehension, where ρ is universally quantified and p a free variable of φ_1. Then, $p \in S$ introduces a new universally quantified formula, i.e., $\rho(p)$, while $p \notin S$ introduces a new existentially quantified formula, i.e., $\neg\rho(p)$. W.l.o.g the existential quantified variables introduces at this step are transformed into free variables, modulo a renaming.

Step 3: instantiate universal quantifiers over the free variables of φ_1. Let $\boldsymbol{p}$ denote the set of free process variables of φ_1 (note that the free variables introduced in Step 2 are not in $\boldsymbol{p}$). Each disjunct $\forall \boldsymbol{y_2}.\, \psi_2$ of φ_2 is equivalently rewritten as $\psi_{2,\exists} \wedge \forall \boldsymbol{y_2}.\, \psi_2$, where $\psi_{2,\exists} = \bigwedge_{\gamma_e \in [\boldsymbol{y_2} \to \boldsymbol{p}]} \psi_2[\gamma_e]$ and $\psi_2[\gamma_e]$ is obtained from ψ_2 by substituting each $y \in \boldsymbol{y_2}$ with $\gamma_e(y)$. Let φ_3 denote the obtained formula.

Step 4: enumerate truth valuations for set and cardinality constraints over free variables in φ_3. Let $\mathcal{A}_s(\varphi_3)$ denote the set of atoms that contain the inclusion or the cardinality operator; each disjunct $\psi_{2,\exists} \wedge \forall \boldsymbol{y_2}.\, \psi_2$ of φ_3 is transformed into the equivalent formula

$$\bigvee_{\gamma_s \in [\mathcal{A}_s(\varphi_3) \to \{0,1\}]} \Big(\bigwedge_{\gamma_s(a)=1} a \wedge \bigwedge_{\gamma_s(a)=0} \neg a \wedge \psi_{2,\exists}[\gamma_s] \wedge \forall \boldsymbol{y_2}.\, \psi_2 \Big), \tag{10}$$

where $\psi_{2,\exists}[\gamma_s]$ is obtained from $\psi_{2,\exists}$ by substituting every $a \in \mathcal{A}_s(\varphi_3)$ with $\gamma_s(a)$.

For each disjunct of the formula in (10), let $\mathcal{C}[\gamma_s]$ be a quantifier-free Presburger formula defined as follows:

• let $\mathcal{T}_s(\varphi_3)$ be the set of terms $S_1 \cap S_2$ or $S_1 \setminus S_2$, where S_1 and S_2 are set variables or applications of function symbols of type $P \to 2^P$ in φ_3 (i.e., they do not contain $\cap$);

• let K_s be a set of integer variables, one variable $k[t]$ for each term $t \in \mathcal{T}_s(\varphi_3)$. Each variable $k[t]$ represents the cardinality of the set denoted by t;

• transform each literal a or $\neg a$ with $a \in \mathcal{A}_s(\varphi_3)$ into a linear constraint over the integer variables K_s:

– if a is a cardinality constraint, then replace every term $|S|$ by the sum of all variables $k[t]$ with $t \in \mathcal{T}_s(\varphi_3)$ of the form $S \cap S'$ or $S \setminus S'$;

– transform set inclusions into cardinality constraints: for every atom a of the form $S_1 \subseteq S_2$, if $\gamma_s(a) = 1$ (resp., $\gamma_s(a) = 0$), a is rewritten as $k[S_2 \setminus S_1] = 0$ (resp., $k[S_1 \setminus S_2] \geq 1$). The extension to more general atoms that use the inclusion operator is straightforward;

• for any atom $p \in S$ from (10) (chosen in Step 2), add $|S| \geq 1$ to $\mathcal{C}[\gamma_s]$; similar constraints can be added for more general constraints of the form $p \in t_S$.

If all Presburger formulas associated with the disjuncts of (10) are unsatisfiable, then φ is unsatisfiable. Otherwise, the formula in (10) is transformed into an equivalent formula of the form

$$\bigvee_{\substack{\gamma_s \in [\mathcal{A}_s(\varphi_3) \to \{0,1\}], \\ \mathcal{C}[\gamma_s] \text{ satisfiable}}} \Big(\psi_{2,\exists}[\gamma_s] \wedge \big(\bigwedge_{\mathcal{C}[\gamma_s] \Rightarrow k[t]>0} \exists p_t.\, p_t \in t \big) \wedge \forall \boldsymbol{y_2}.\, \psi_2 \Big) \tag{11}$$

Step 5: Note that all $\psi_{2,\exists}[\gamma_s]$ are quantifier-free formulas with uninterpreted functions and order constraints, for which the satisfiability problem is decidable.

If all $\psi_{2,\exists}[\gamma_s]$ are unsatisfiable then φ is unsatisfiable, otherwise the procedure returns to Step 1 by letting φ be the disjunction of all formulas in (11) that have a satisfiable $\psi_{2,\exists}[\gamma_s]$ sub-formula.

5.3 Completeness

We identify a fragment of $\mathbb{CL}$, denoted $\mathbb{CL}_{dec}$, whose satisfiability problem is decidable. The proof is based on a small model argument. The syntactical restrictions in $\mathbb{CL}_{dec}$ are: (1) function symbols of type $P \rightarrow P$ are used only in constraints on data, i.e., they occur only in data terms of the form $\mathtt{x}(f(p))$, (2) there exists no atomic formula that contains two different function symbols of type $P \rightarrow 2^P$, (3) cardinality constraints are restricted to atomic formulas of the form $\varphi_{\mathbb{Z}} ::= c * |t_s| \geq exp \ \mid\ |t_s^1| \geq |t_s^2| \ \mid\ exp = c * n + b$, with $b, c \in \mathbb{Z}$, and (4) set comprehensions are defined using conjunctions of quantifier-free atomic formulas or universally quantified formulas of the form:

$$S = \{q \mid \forall \boldsymbol{t}.\, \varphi_P(q, \boldsymbol{t})\} \mid S = \{q \mid \forall \boldsymbol{t}.\, \varphi_{\mathbb{Z}}(q, \boldsymbol{t})\} \mid S = \{q \mid \forall \boldsymbol{t}.\, \boldsymbol{t} \in F(q) \Rightarrow \varphi^{loc}(q, \boldsymbol{t})\}$$
$$\text{with } \varphi^{loc}(q, \boldsymbol{t}) ::= \varphi_D(q, \boldsymbol{t}) \mid \varphi_{\mathbb{Z}}(q, \boldsymbol{t}) \mid \varphi^{loc}(q, \boldsymbol{t}) \wedge \varphi^{loc}(q, \boldsymbol{t}) \mid \neg \varphi^{loc}(q, \boldsymbol{t})$$

where the $\boldsymbol{t}$, q appear only in terms of the form $F[\Pi](q)$, $\mathtt{x}(q)$, $\mathtt{x}(f(q))$, $\mathtt{x}(t)$ with Π an atomic data formula over $\boldsymbol{t}$ of q.

Let S be a set comprehension whose definition uses universal quantification and terms of the form $F(t)$, with t universally quantified. The set S defines a relation between its elements and a potentially unbounded number of processes in the network. They are called *relational set comprehensions*. An example is the kernel of a network, i.e., $K = \{q \mid \forall t.\, q \in HO(t)\}$. In $\mathbb{CL}_{dec}$, the only constraints allowed on relational set comprehensions are lower bounds on their cardinality, i.e., given such a set S, it occurs only in atomic formulas $|S| \geq exp$ or $|S| \geq |t_S|$ under an even number of negations.

The uncoordinated algorithms in [10] are captured by $\mathbb{CL}_{dec}$, including our running example.

Theorem 1. *The satisfiability problem for formulas in* $\mathbb{CL}_{dec}$ *is decidable.*

Proof (Sketch): The small model property for this fragment of $\mathbb{CL}$ can be stated as follows: for any formula φ, if φ is satisfiable then φ has a model whose size is bounded by a minimal solution of a quantifier-free Presburger formula constructed from φ; the order relation on solutions, i.e., on tuples of integers, is defined component-wise. Note that, in general, there are exponentially-many minimal solutions for a quantifier-free Presburger formula.

The Presburger formula is constructed from φ by applying a modified version of Step 4 from the semi-decision procedure in Sec. 5.2. One starts by considering the Venn diagram induced by the set/process variables used in the formula (process variables are considered singleton sets) and enumerating all possibilities of a region to be empty or not. For each non-empty region and each function symbol of type $P \rightarrow P$ (resp., $P \rightarrow 2^P$), we introduce a fresh process variable (resp., set variable) representing the value of this function for all the elements in this region. This is possible because it can be proven that if φ has a model of size n then it has also a model of the same size, where all the nodes in the same Venn region share the same value for their function symbols. Then, one enumerates all truth valuations for the cardinality constraints and constructs a Presburger formula encoding these constraints over a larger (exponential) set of integer variables, one for each region of the Venn diagram (this diagram includes also the set/process variables introduced to denote values of function symbols).

Given a bound on the small models, one can enumerate all network graphs of size smaller than this bound, and compute a quantifier-free formula with uninterpreted functions and order constraints for each one of them. The original formula is satisfiable iff there exists such a quantifier-free formula which is satisfiable.

5.4 Discussion

The semi-decision procedure introduced in Sec. 5.2 is targeting the specific class of verification conditions for consensus. Intuitively, when designing consensus algorithms one wants to avoid that two disjunct sets of processes decide independently of each other, as this may lead to a violation of agreement. There are two ways to avoid it. First, the algorithm can use a topology predicate to enforce that any two HO-sets intersect (no-split). Second, the algorithm can ensure that a decision is made only if "supported" by a *majority* of processes. When we apply the semi-decision procedure on formulas expressing the negation of these two statements, typically it proves them unsatisfiable. It derives a contradiction starting from the assumption that two sets are disjoint due to their definition (by comprehension). In the first case, the contradiction is obtained by exploiting an explicit cardinality constraint on the intersection, i.e., that the cardinality of the intersection is greater than or equal to 1. In the second case, the contradiction is derived from the fact that each of the two sets have cardinality greater than $n/2$ (majority). For this, one needs to enumerate all pairs of sets and check that their cardinality constraints imply non-empty intersection.

For arbitrary formulas in $\mathbb{CL}$ our semi-decision procedure may fail to derive a contradiction, because one may need to explore the exponentially many regions of the Venn diagram that are induced by the sets represented in the formula. For the decidable fragment $\mathbb{CL}_{dec}$, this is done by the decision procedure in Sec. 5.3.

To conclude, our semi-decision procedure targets the specific class of verification conditions needed for consensus. The semi-decision procedure proves the unsatisfiability of formulas that are not in $\mathbb{CL}_{dec}$. Compared to the decision procedure for $\mathbb{CL}_{dec}$, the semi-decision procedure avoids deciding quantifier-free Presburger formulas over an exponential number of variables and computing all minimal solutions of such formulas (which are exponentially many).

6 Evaluation

We have evaluated our framework on several fault-tolerant consensus algorithms taken from [10], [4], and [19]. All algorithms in [10] and [4] fit into our frame-

Table 1. Experimental results. (1) coordinated, (2) number of rounds per phase, (3) number of invariants provided by the user (safety + termination), (4) number of verification conditions, (5) total solving time.

Algorithm	coord. (1)	rounds (2)	invariants (3)	VCs (4)	solving (5)
Uniform Voting [10]	$\times$	2	2+2	13	< 0.1s
Coordinated Uniform Voting [10]	✓	3	2+2	10	< 0.1s
Simplified Coordinated Uniform Voting [10]	✓	2	2+2	8	< 0.1s
One Third Rule [10]	$\times$	1	1+1	8	< 0.1s
Last Voting [10]	✓	4	1+3	15	1s
$\mathcal{A}_{T,E}$ [4]	$\times$	1	1+2	10	< 0.1s
$\mathcal{U}_{T,E}$ [4]	$\times$	2	2+2	9	< 0.1s
FloodMin [19, Chapter 6.2.1]	$\times$	1	1	5	< 0.1s

work. We tested our semi-decision procedure by manually encoding the algorithms, invariants, and properties in the SMT-LIB 2 format and used the Z3 [21] SMT-solver to check satisfiability of the formulas produced by the semi-decision procedure. In the reduction, we inline the minimization problem along the rest of the formula and let Z3 instantiate the universal quantifiers. The results are summarized in Table 1. The files containing the verification conditions are available at `http://pub.ist.ac.at/~zufferey/consensus/`. We give a short description of each algorithm and how it is proven correct in our framework. The consensus algorithms we considered are presented in a way such that several consecutive rounds are grouped together into a phase. This is done, because the computation transition is different for each round within a phase. We verified agreement, validity, irrevocability, and termination.

The *Uniform Voting* algorithm is a deterministic version of the Ben-Or randomized algorithm [3]. The condition for safety is that all environment transitions satisfy the topology predicate $\forall i, j.\, |HO(i) \cap HO(j)| \geq 1$, called *no-split*. Intuitively, a process decides a value v if all the messages it has received in a specific round are v. Thus two processes decide on different values only if their HO-sets are disjoint. Roughly, the semi-decision procedure succeeds in finding a contradiction, by exploring the explicit non-empty intersection of HO-sets defined by the topology predicate; more specifically the non-empty intersection of HO-sets of two processes that are supposed to decide differently.

Coordinated Uniform Voting and *simplified Coordinated Uniform Voting* [10] are coordinated algorithms. Reasoning about topology predicates similar to *Uniform Voting* leads to safety of these two algorithms. In fact *simplified Coordinated Uniform Voting* is based on an even stronger topology predicate than no-split.

The *One Third Rule* algorithm is our running example. It is designed to be safe without any topology predicate. In this case, the computation transitions enforce that if a processes decides, a majority of processes are in a specific state. In Sec. 5.2, the overview explains how the semi-decision procedure derives a contradiction to prove one of the verification conditions for safety using the majority argument. Our proof of termination is based on a stronger communication predicate than the one provided in [10], namely, it requires two uniform rounds where more that 2/3 of the messages are received by each process.

The *Last Voting* algorithm is an encoding of Paxos [17] in the HO-model. This algorithm is coordinated. Similar to One Third Rule it is safe without any topology predicate, and the algorithm imposes cardinality constraints that create majority sets: Before voting or deciding, the coordinator makes sure that a majority of process acknowledged its messages, such that a decision on v implies $|\{p \,|\, \mathtt{x}(p) = v\}| > n/2$.

The $\mathcal{A}_{T,E}$ algorithm [4] is a generalization of the One Third Rule to value faults that uses different thresholds. It tolerates less than $n/4$ corrupted messages per process. Safety and termination of the algorithm follows the same type of reasoning as the One Third Rule algorithm but require more complex reasoning about the messages. To model value faults, the *HO*-sets are partitioned into a safe part (*SHO*) and an altered part (*AHO*). A message from process p to process q is discarded if $p \notin HO(q)$, delivered as expected if $p \in SHO(q)$, and if $p \in AHO(q)$ an arbitrary message is delivered instead of the original one. The $\mathcal{U}_{T,E}$ algorithm [4] is an consensus algorithm with value faults designed for communication which is live but not safe. For $\mathcal{A}_{T,E}$ and $\mathcal{U}_{T,E}$, to simplify the manually encoding, we have considered the intersection of up to three sets rather than two as presented in the semi-decision procedure.

The *FloodMin* algorithm is a synchronous consensus algorithm tolerating at most f crash fault [19, Chapter 6.2.1]. In each round the processes sends their value to all the processes and keep the smallest received value. Executing $f + 1$ rounds guarantees that there is at least one round where no process crashes. Agreement is reached in this (special) round. The invariant captures that fact by counting the number of crashed process and relating it to the number of processes with different values, i.e. $|C| < r \Rightarrow \exists v. \, |\{p \mid \mathtt{x}(p) = v\}| = n$. Proving termination requires a ranking function.

7 Related Work

The verification of distributed algorithms is a very challenging task. Indeed, most of the verification problems are undecidable for parameterized systems [2,23]. Infinite-state model-checking techniques may be applied if one restricts the type of actions performed by the processes. Particular classes of systems which are monotonic enjoy good decidability properties [1,13]. Fault-tolerant distributed algorithms cannot be modeled as such restricted systems. Recently, John et al. [15] developed abstractions suitable for model-checking threshold-based fault-tolerant distributed algorithms.

Orthogonally to the model-checking approach and closer to our approach is the formalization of programs and their specifications in logics where the satisfiability question is decidable. Very expressive logics have been explored for the verification of data structures and $\mathbb{CL}$ is a new combination of the constructs present in those logics. The array property fragment [6] admits a limited form of quantifier alternation which is close to ours. Reasoning about sets and cardinality constraints is present in BAPA [16]. However, BAPA does not combine well with function symbols over sets [24]. Logics for linked heap structures such as lists are similar to $\mathbb{CL}$ if we encode sets as lists and cardinality constraints as length constraints. STRAND [20] and CSL [5] offer more quantifier alternations and richer constraints on data but have more limited cardinality constraints. Both logics have decision procedures based on a small model property.

If one accepts less automation, distributed algorithms can be formalized and verified in interactive proof assistants. For instance, Isabelle has been used to verify algorithms in the heard-of model [9]. The verification of distributed systems has also been tackled using the TLA+ specification language [18].

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems (1996)
2. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 15, 307–309 (1986)
3. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: PODC, pp. 27–30. ACM (1983)
4. Biely, M., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper, A., Widder, J.: Tolerating corrupted communication. In: PODC, pp. 244–253 (2007)
5. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
6. 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)
7. Brasileiro, F., Greve, F.G.P., Mostéfaoui, A., Raynal, M.: Consensus in one communication step. In: Malyshkin, V.E. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 42–50. Springer, Heidelberg (2001)
8. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: OSDI. USENIX Association, Berkeley (2006)
9. Charron-Bost, B., Merz, S.: Formal verification of a consensus algorithm in the heard-of model. Int. J. Software and Informatics 3(2-3), 273–303 (2009)
10. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
12. Függer, M., Schmid, U.: Reconciling fault-tolerant distributed computing and systems-on-chip. Dist. Comp. 24(6), 323–355 (2012)
13. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. Journal of the ACM 39, 675–735 (1992)
14. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: USENIXATC. USENIX Association (2010)
15. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD, pp. 201–209 (2013)
16. Kuncak, V., Nguyen, H.H., Rinard, M.: An algorithm for deciding BAPA: Boolean algebra with presburger arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 260–277. Springer, Heidelberg (2005)

17. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. (1998)
18. Lamport, L.: Distributed algorithms in TLA (abstract). In: PODC (2000)
19. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
20. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622. ACM (2011)
21. 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)
22. Santoro, N., Widmayer, P.: Time is not a healer. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–313. Springer, Heidelberg (1989)
23. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. 28(4), 213–214 (1988)
24. Yessenov, K., Piskac, R., Kuncak, V.: Collections, cardinalities, and relations. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 380–395. Springer, Heidelberg (2010)

Verifying Array Programs by Transforming Verification Conditions

Emanuele De Angelis[1], Fabio Fioravanti[1], Alberto Pettorossi[2], and Maurizio Proietti[3]

[1] DEC, University 'G. D'Annunzio', Pescara, Italy
`{emanuele.deangelis,fioravanti}@unich.it`
[2] DICII, University of Rome Tor Vergata, Rome, Italy
`pettorossi@disp.uniroma2.it`
[3] IASI-CNR, Rome, Italy
`maurizio.proietti@iasi.cnr.it`

Abstract. We present a method for verifying properties of imperative programs manipulating integer arrays. We assume that we are given a program and a property to be verified. The *interpreter* (that is, the operational semantics) of the program is specified as a set of Horn clauses with constraints in the domain of integer arrays, also called *constraint logic programs over integer arrays*, denoted CLP(Array). Then, by specializing the interpreter with respect to the given program and property, we generate a set of *verification conditions* (expressed as a CLP(Array) program) whose satisfiability implies that the program verifies the given property. Our verification method is based on transformations that preserve the least model semantics of CLP(Array) programs, and hence the satisfiability of the verification conditions. In particular, we apply the usual rules for CLP transformation, such as unfolding, folding, and constraint replacement, tailored to the specific domain of integer arrays. We propose an automatic strategy that guides the application of those rules with the objective of deriving a new set of verification conditions which is either trivially satisfiable (because it contains no constrained facts) or is trivially unsatisfiable (because it contains the fact *false*). Our approach provides a very rich program verification framework where one can compose together several verification strategies, each of them being implemented by transformations of CLP(Array) programs.

1 Introduction

Horn clauses and constraints have been advocated by many researchers as suitable logical formalisms for the automated verification of imperative programs [2,19,34]. Indeed, the *verification conditions* that express the correctness of a given program, can often be expressed as *constrained Horn clauses* [3], that is, Horn clauses extended with constraints in specific domains such as the integers or the rationals. For instance, consider the following C-like program *prog*:

$x\!=\!0;\ \ y\!=\!0;$
`while` $(x\!<\!n)\ \{x\!=\!x\!+\!1;\ \ y\!=\!y\!+\!2\}$

and assume that we want to prove the following Hoare triple: $\{n \geq 1\}$ *prog* $\{y > x\}$. This triple is valid if we find a predicate P such that the following three verification conditions hold:

1. $x = 0 \land y = 0 \land n \geq 1 \;\rightarrow\; P(x, y, n)$
2. $P(x, y, n) \land x < n \;\rightarrow\; P(x + 1, y + 2, n)$
3. $P(x, y, n) \land x \geq n \;\rightarrow\; y > x$

Constraints such as the equalities and inequalities in clauses 1–3, are formulas defined in a background (possibly non-Horn) theory. The use of constraints makes it easier to express the properties of interest and enables us to apply ad-hoc theorem provers, or *solvers*, for reasoning over those properties.

Verification conditions can be automatically generated either from a formal specification of the operational semantics of the programs [34] or from the proof rules that formalize program correctness in an axiomatic way [19].

The correctness of a program is implied by the satisfiability of the verification conditions. Various methods and tools for *Satisfiability Modulo Theory* (see, for instance, [11]) prove the correctness of a given program by finding an interpretation (that is, a relation specified by constraints) that makes the verification conditions true. For instance, in our example, one such interpretation is:

$$P(x, y, n) \;\equiv\; (x = 0 \land y = 0 \land n \geq 1) \lor y > x$$

It has been noted (see, for instance, [3]) that verification conditions can be viewed as *constraint logic programs*, also called *CLP programs* [22]. Indeed, clauses 1 and 2 above can be considered as clauses of a CLP program over the integers, and clause 3 can be rewritten as the following *goal* (by moving the conclusion to the premises):

4. $P(x, y, n) \land x \geq n \land y \leq x \;\rightarrow\; \mathit{false}$

Various verification methods based on constraint logic programming have been proposed in the literature (see, for instance, [8,10,34]). These methods consist of two steps: (i) the first step is the translation of the verification task into a CLP program, and (ii) the second step is the analysis of that CLP program. In particular, as indicated in [8], in many cases it is helpful for the analysis step to transform a CLP program (expressing a set of verification conditions) into an equisatisfiable program whose satisfiability is easier to show.

For instance, if we propagate, according to the transformations described in [8], the two constraints representing the initialization condition ($x = 0 \land y = 0 \land n \geq 1$) and the error condition ($x \geq n \land y \leq x$), then from clauses 1, 2, and 4 we derive the following new verification conditions:

5. $Q(x, y, n) \land x < n \land x > y \land y \geq 0 \;\rightarrow\; Q(x + 1, y + 2, n)$
6. $Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \;\rightarrow\; \mathit{false}$

This propagation of constraints preserves the least model, and hence, by extending the van Emden-Kowalski Theorem [38] to constrained Horn clauses, the verification conditions expressed by clauses 5–6 are satisfiable iff clauses 1–3 are satisfiable. Now, proving the satisfiability of clauses 5–6 is trivial because none of them is a *constrained fact* (that is, a clause of the form $c \rightarrow Q(x, y, n)$, where c is a satisfiable constraint). Thus, clauses 5-6 are made true by simply taking $Q(x, y, n)$ to be *false*.

The approach presented in [8] shows that the transformational verification method briefly presented in the example above, is quite general. According to that method, in fact, one starts from a program *prog* on integers and a safety property φ to be verified. Then, following [34], one specifies the *interpreter* of the program as a CLP program whose constraints are in the domain of integer arrays. Next, by specializing the interpreter with respect to *prog* and φ, a new CLP program, call it *VC*, is derived. This program consists of the clauses that express the verification conditions (hence the name *VC*) which guarantee that *prog* satisfies φ. Program *VC* (and the corresponding set of verification conditions) is repeatedly specialized with respect to the constraints occurring in its clauses with the objective of deriving either (i) a CLP program without constrained facts, hence proving that *prog* satisfies φ, or (ii) a CLP program containing the fact *false*, hence proving that *prog* does not satisfy φ (in this case a counterexample to φ can be extracted from the derivation of the specialized program).

In this paper we extend the method presented in [8] to the proof of partial correctness properties of programs manipulating integer arrays. In order to specify verification conditions for array programs, in Section 2 we introduce the class of CLP(Array) programs, that is, logic programs with constraints in the domain of integer arrays. In particular, CLP(Array) programs may contain occurrences of `read` and `write` predicates that are interpreted as the input and output relations of the usual read and write operations on arrays. Then, in Section 3 we introduce some transformation rules for manipulating CLP(Array) programs. Besides the usual *unfolding* and *folding* rules, we consider the *constraint replacement* rule, which allows us to replace constraints by equivalent ones in the theory of arrays [4,17,30]. In Section 4 we show how to generate the verification conditions via specialization of CLP(Array) programs. In Section 5 we present an automatic strategy designed for applying the transformation rules with the objective of obtaining a proof (or a disproof) of the properties of interest. In particular, similarly to [8], the strategy aims at deriving either (i) a CLP(Array) program that has no constrained facts (hence proving satisfiability of the verification conditions and partial correctness of the program), or (ii) a CLP(Array) program containing the fact *false* (hence proving that the verification conditions are unsatisfiable and the program does not satisfy the given property). The transformation strategy may introduce some auxiliary predicates by using a *generalization strategy* that extends to CLP(Array) the generalization strategies for CLP programs over integers or reals [14]. Finally, as reported in Section 6, we have implemented our transformation strategy on the MAP transformation system [29] and we have tested the verification method using the strategy we have proposed on a set of array programs taken from the literature.

2 Constraint Logic Programs on Arrays

In this section we recall some basic notions and terminology concerning Constraint Logic Programming (CLP), and we introduce the set CLP(Array) of CLP programs with constraints in the domain of integer arrays. For details on CLP the reader may refer to [22].

If $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer variables and coefficients, then $\mathtt{p}_1 = \mathtt{p}_2$, $\mathtt{p}_1 \geq \mathtt{p}_2$, and $\mathtt{p}_1 > \mathtt{p}_2$ are *atomic integer constraints*. The dimension n of an array a is represented as a binary relation by the predicate $\mathtt{dim}(\mathtt{a},\mathtt{n})$. For reasons of simplicity we consider one-dimensional arrays only. The read and write operations on arrays are represented by the predicates read and write, respectively, as follows: $\mathtt{read}(\mathtt{a},\mathtt{i},\mathtt{v})$ denotes that the i-th element of array a is the value v, and $\mathtt{write}(\mathtt{a},\mathtt{i},\mathtt{v},\mathtt{b})$ denotes that the array b that is equal to the array a except that its i-th element is v. We assume that both indexes and values are integers, but our method is parametric with respect to the index and value domains. (Note, however, that the result of a verification task may depend on the constraint solver used, and hence on the constraint domain.)

An *atomic array constraint* is an atom of the following form: either $\mathtt{dim}(\mathtt{a},\mathtt{n})$, or $\mathtt{read}(\mathtt{a},\mathtt{i},\mathtt{v})$, or $\mathtt{write}(\mathtt{a},\mathtt{i},\mathtt{v},\mathtt{b})$. A *constraint* is either true, or an atomic (integer or array) constraint, or a *conjunction* of constraints. An *atom* is an atomic formula of the form $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{m})$, where p is a predicate symbol not in $\{=, \geq, >, \mathtt{dim}, \mathtt{read}, \mathtt{write}\}$ and $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m}$ are terms constructed out of variables, constants, and function symbols different from + and *.

A CLP(Array) program is a finite set of clauses of the form `A :- c, B`, where A is an atom, c is a constraint, and B is a (possibly empty) conjunction of atoms. A is called the *head* and `c, B` is called the *body* of the clause. We assume that in every clause all integer arguments in its head are distinct variables. The clause `A :- c` is called a *constrained fact*. When c is true then it is omitted and the constrained fact is called a *fact*. A *goal* is a formula of the form `:- c, B` (standing for $c \wedge B \rightarrow \mathit{false}$ or, equivalently, $\neg(c \wedge B)$). A CLP(Array) program is said to be *linear* if all its clauses are of the form `A :- c, B`, where B consists of at most one atom.

We say that a predicate p *depends on* a predicate q in a program P if either in P there is a clause of the form `p(...) :- c, B` such that q occurs in B, or there exists a predicate r such that p depends on r in P and r depends on q in P. We say that a predicate p in a linear program P is *useless* if in P there are constrained facts neither for p nor for each predicate q on which q depends.

Now we define the semantics of CLP(Array) programs. An $\mathcal{A}$-*interpretation* is an interpretation I, that is, a set D, a function in $D^n \rightarrow D$ for each function symbol of arity n, and a relation on D^n for each predicate symbol of arity n, such that:

(i) the set D is the Herbrand universe [28] constructed out of the set $\mathbb{Z}$ of the integers, the constants, and the function symbols different from + and *,
(ii) I assigns to $+, *, =, \geq, >$ the usual meaning in $\mathbb{Z}$,
(iii) for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$, for all integers d,
$\mathtt{dim}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{d})$ is true in I iff $\mathtt{d} = \mathtt{n}$

(iv) I interprets the predicates `read` and `write` as follows: for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$ and $\mathtt{b}_0 \ldots \mathtt{b}_{\mathtt{m}-1}$ of integers, for all integers $\mathtt{i}$ and $\mathtt{v}$,
$\mathtt{read}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{i}, \mathtt{v})$ is true in I iff $0 \le \mathtt{i} \le \mathtt{n}-1$ and $\mathtt{v} = \mathtt{a}_\mathtt{i}$, and
$\mathtt{write}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{i}, \mathtt{v}, \mathtt{b}_0 \ldots \mathtt{b}_{\mathtt{m}-1})$ is true in I iff
$0 \le \mathtt{i} \le \mathtt{n}-1$, $\mathtt{n} = \mathtt{m}$, $\mathtt{b}_\mathtt{i} = \mathtt{v}$, and for $\mathtt{j} = 0, \ldots, \mathtt{n}-1$, if $\mathtt{j} \ne \mathtt{i}$ then $\mathtt{a}_\mathtt{j} = \mathtt{b}_\mathtt{j}$
(v) I is an Herbrand interpretation [28] for function and predicate symbols different from $+, *, =, \ge, >$, `dim`, `read`, and `write`.

We can identify an $\mathcal{A}$-interpretation I with the set of ground atoms that are true in I, and hence $\mathcal{A}$-interpretations are partially ordered by set inclusion.

We write $\mathcal{A} \models \varphi$ if φ is true in every $\mathcal{A}$-interpretation. A constraint `c` is *satisfiable* if $\mathcal{A} \models \exists(\mathtt{c})$, where in general, for every formula φ, $\exists(\varphi)$ denotes the existential closure of φ. Likewise, $\forall(\varphi)$ denotes the universal closure of φ. A constraint is *unsatisfiable* if it is not satisfiable. A constraint `c` *entails* a constraint `d`, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$. By $vars(\varphi)$ we denote the free variables of φ.

We assume that we are given a solver to check the satisfiability and the entailment of constrains in $\mathcal{A}$. To this aim we can use any solver that implements algorithms for satisfiability and entailment in the theory of integer arrays [4,17].

The semantics of a CLP(Array) program P is defined to be the *least $\mathcal{A}$-model* of P, denoted $M(P)$, that is, the least $\mathcal{A}$-interpretation I such that every clause of P is true in I.

Given a CLP(Array) program P and a ground goal `G` of the form `:-A`, $P \cup \{\mathtt{G}\}$ is satisfiable (or, equivalently, $P \not\models \mathtt{A}$) if and only if $\mathtt{A} \notin M(P)$. This property is a straightforward extension to CLP(Array) programs of van Emden and Kowalski's result [38].

3 Transformation Rules for CLP(Array) Programs

Our verification method is based on the application of transformations that, under suitable conditions, preserve the least $\mathcal{A}$-model semantics of CLP(Array) programs. In particular, we apply the following *transformation rules*, collectively called *unfold/fold rules*: (i) *definition*, (ii) *unfolding*, (iii) *constraint replacement*, and (iv) *folding*. These rules are an adaptation to CLP(Array) programs of the unfold/fold rules for a generic CLP language (see, for instance, [13]).

Let P be a CLP(Array) program.

Definition Rule. By this rule we introduce a clause of the form `newp(X) :- c,A`, where `newp` is a new predicate symbol (occurring neither in P nor in a clause introduced by the definition rule), `X` is the tuple of variables occurring in the atom `A`, and `c` is a constraint.

Unfolding Rule. Given a clause C of the form `H :- c,L,A,R`, where `H` and `A` are atoms, `c` is a constraint, and `L` and `R` are (possibly empty) conjunctions of atoms, let us consider the set $\{\mathtt{K}_i \text{ :- } \mathtt{c}_i, \mathtt{B}_i \mid i = 1, \ldots, m\}$ made out of the (renamed apart) clauses of P such that, for $i = 1, \ldots, m$, `A` is unifiable with $\mathtt{K}_i$ via the most general unifier ϑ_i and $(\mathtt{c}, \mathtt{c}_i)\,\vartheta_i$ is satisfiable. By unfolding C w.r.t. A using P, we derive the set $\{(\mathtt{H} \text{ :- } \mathtt{c}, \mathtt{c}_i, \mathtt{L}, \mathtt{B}_i, \mathtt{R})\,\vartheta_i \mid i = 1, \ldots, m\}$ of clauses.

Constraint Replacement Rule. If a constraint $\mathtt{c}_0$ occurs in the body of a clause C and, for some constraints $\mathtt{c}_1, \ldots, \mathtt{c}_\mathtt{n}$,

$\mathcal{A} \models \forall\,((\exists \mathtt{X}_0\, \mathtt{c}_0) \leftrightarrow (\exists \mathtt{X}_1\, \mathtt{c}_1 \vee \ldots \vee \exists \mathtt{X}_\mathtt{n}\, \mathtt{c}_\mathtt{n}))$

where, for $i = 0, \ldots, n$, $\mathtt{X}_\mathtt{i} = \mathit{vars}(C) - \mathit{vars}(\mathtt{c}_\mathtt{i})$, then we derive n new clauses $C_1, \ldots, C_n$ by replacing $\mathtt{c}_0$ by $\mathtt{c}_1, \ldots, \mathtt{c}_\mathtt{n}$, respectively, in the body of C.

The equivalences needed for constraint replacements are shown to hold in $\mathcal{A}$ by using a relational version of the theory of arrays with dimension [4,17]. In particular, the constraint replacements we apply during the transformations described in Section 5 follow from the following axioms where all variables are universally quantified at the front:

(A1) $\mathtt{I = J,\ read(A, I, U),\ read(A, J, V)} \rightarrow \mathtt{U = V}$

(A2) $\mathtt{I = J,\ write(A, I, U, B),\ read(B, J, V)} \rightarrow \mathtt{U = V}$

(A3) $\mathtt{I \neq J,\ write(A, I, U, B),\ read(B, J, V)} \rightarrow \mathtt{read(A, J, V)}$

Axiom (A1) is often called *array congruence* and axioms (A2) and (A3) are collectively called *read-over-write*. We omit the usual axioms for `dim`.

Folding Rule. Given a clause E: `H :- e, L, A, R` and a clause D: `K :- d, D` introduced by the definition rule. Suppose that, for some substitution ϑ, (i) $\mathtt{A} = \mathtt{D}\,\vartheta$, and (ii) $\forall\,(\mathtt{e} \rightarrow \mathtt{d}\,\vartheta)$. Then by folding E using D we derive $\mathtt{H :- e, L, K}\,\vartheta\mathtt{, R}$.

From P we can derive a new program *TransfP* by: (i) selecting a clause C in P, (ii) deriving a new set *TransfC* of clauses by applying one or more transformation rules, and (iii) replacing C by *TransfC* in P. Clearly, we can apply a new sequence of transformation rules starting from *TransfP* and iterate this process at will.

The correctness results for the unfold/fold transformations of CLP programs proved in [13] can be instantiated to our context as stated in the following theorem.

Theorem 1. (Correctness of the Transformation Rules) *Let the CLP(Array) program TransfP be derived from P by a sequence of applications of the transformation rules. Suppose that every clause introduced by the definition rule is unfolded at least once in this sequence. Then, for every ground atom A in the language of P, $A \in M(P)$ iff $A \in M(\mathit{TransfP})$.*

The assumption that the unfolding rule should be applied at least once is required for technical reasons (see the details in [13]). Informally, this assumption avoids the replacement of a definition clause `A :- B` with the clause `A :- A` obtained by folding `A :- B` using itself. This replacement may not preserve the least model semantics.

4 Generating Verification Conditions via Specialization

We consider an imperative C-like programming language with integer and array variables, assignments (`=`), sequential compositions (`;`), conditionals (`if else`), while-loops (`while`), and jumps (`goto`). A program is a sequence of (labeled) commands, and in each program there is a unique `halt` command which, when executed, causes program termination.

The semantics of our language is defined by a *transition relation*, denoted $\Longrightarrow$, between *configurations*. Each configuration is a pair $\langle\!\langle c, \delta \rangle\!\rangle$ of a command c and an *environment* δ. An environment δ is a function that maps: (i) every integer variable identifier x to its value v, and (ii) every integer array identifier a to a *finite* sequence $\mathtt{a}_0, \ldots, \mathtt{a}_{\mathtt{n}-1}$ of integers, where $\mathtt{n}$ is the dimension of the array a. The definition of the relation $\Longrightarrow$ is similar to the 'small step' operational semantics given in [36], and is omitted.

Given an imperative program *prog*, we address the problem of verifying whether or not, starting from any *initial configuration* that satisfies the property φ_{init}, the execution of *prog* eventually leads to a *final configuration* that satisfies the property φ_{error}, also called an *error configuration*. This problem is formalized by defining an *incorrectness triple* of the form $\{\!\{\varphi_{init}\}\!\}$ *prog* $\{\!\{\varphi_{error}\}\!\}$, where φ_{init} and φ_{error} are constraints. We say that a program *prog* is *incorrect* with respect to φ_{init} and φ_{error}, whose free variables are assumed to be among the program variables $z_1, \ldots, z_r$, if there exist environments δ_{init} and δ_h such that: (i) $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$ holds, (ii) $\langle\!\langle \ell_0 : c_0,\ \delta_{init} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \mathtt{halt},\ \delta_h \rangle\!\rangle$, and (iii) $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$ holds, where $\ell_0 : c_0$ is the first labeled command of *prog* and $\ell_h : \mathtt{halt}$ is the unique `halt` command of *prog*. A program is said to be *correct* with respect to φ_{init} and φ_{error} iff it is not incorrect with respect to φ_{init} and φ_{error}. Note that our notion of correctness is equivalent to the usual notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$. In this paper we assume that the properties φ_{init} and φ_{error} can be expressed as conjunctions of (integer and array) constraints.

We translate the problem of checking whether or not the program *prog* is incorrect with respect to the properties φ_{init} and φ_{error} into the problem of checking whether or not the nullary predicate `incorrect` (standing for *false*) is a consequence of the CLP(Array) program T defined by the following clauses:

```
incorrect :- errorConf(X), reach(X).
reach(Y) :- tr(X,Y), reach(X).
reach(Y) :- initConf(Y).
```

together with the clauses for the predicates `initConf(X)`, `errorConf(X)`, and `tr(X,Y)`. Those clauses are defined as follows: (i) `initConf(X)` encodes an initial configuration satisfying the property φ_{init}, (ii) `errorConf(X)` encodes an error configuration satisfying the property φ_{error}, and (iii) `tr(X,Y)` encodes the transition relation $\Longrightarrow$ between pairs of configurations, which depends on the given program *prog*. For instance, the following clause encodes the transition relation for the array assignment $\ell : a[ie] = e$ (here a configuration pair of the form: $\langle\!\langle \ell : c, \delta \rangle\!\rangle$ for the command c at label ℓ and the environment δ, is denoted by the term `cf(cmd(L,C),D)`):

```
tr(cf(cmd(L,asgn(arrayelem(A,IE),E)),D), cf(cmd(L1,C),D1)) :-
    eval(IE,D,I), eval(E,D,V), lookup(D,array(A),FA), write(FA,I,V,FA1),
    update(D,array(A),FA1,D1), nextlab(L,L1), at(L1,C).
```

(`L1` is the label following `L` in the encoding of the given program.) The predicate `reach(Y)` holds if a configuration `Y` can be reached from an initial configuration.

The imperative program *prog* is correct with respect to the properties φ_{init} and φ_{error} iff `incorrect`$\notin M(T)$ (or, equivalently, $T \not\models$ `incorrect`), where $M(T)$ is the least $\mathcal{A}$-model of program T (see Section 2). Our verification method performs a sequence of applications of the unfold/fold rules presented in Section 3 starting from program T. By Theorem 1 we have that, for each program U obtained from T by a sequence of applications of the rules, `incorrect`$\in M(T)$ iff `incorrect`$\in M(U)$.

Our verification method is made out of the following two steps, each of which is realized by a sequence of applications of the unfold/fold transformation rules: *Step* (A): *Generation of Verification Conditions*, and *Step* (B): *Transformation of Verification Conditions.*

In Step (A) program T is *specialized* with respect to the given `tr` (which depends on *prog*), `initConf`, and `errorConf`, thereby deriving a new program $T1$ such that: (i) `incorrect` $\in M(T)$ iff `incorrect` $\in M(T1)$, and (ii) `tr` does not occur explicitly in $T1$. The specialization of T is obtained by applying a variant of the strategy for interpreter removal presented in [8]. The main difference with respect to [8] is that the CLP programs considered in this paper contain `read`, `write`, and `dim` predicates. The `read` and `write` predicates are never unfolded during specialization and they occur in the residual CLP(Array) program $T1$. All occurrences of the `dim` predicate are eliminated by replacing them by suitable integer constraints on indexes. The clauses of $T1$ are called the *verification conditions* for *prog*, and we say that they are *satisfiable* iff `incorrect` $\notin M(T1)$ (or equivalently $T1 \not\models$ `incorrect`). Thus, the satisfiability of the verification conditions for *prog* guarantees that *prog* is correct with respect to φ_{init} and φ_{error}.

Step (B) has the objective of checking, through further transformations, the satisfiability of the verification conditions generated by Step (A). We will describe this step in detail in Section 5.

Let us consider, for example, the following program *SeqInit* which initializes a given array a of n integers by the sequence: $a[0],\ a[0]+1,\ \ldots,\ a[0]+n-1$:

```
SeqInit:     $\ell_0$: $i = 1$;
             $\ell_1$: while ($i<n$) { $a[i] = a[i-1] + 1$; $i = i + 1$; };
             $\ell_h$: halt
```

We consider the following incorrectness triple:

$$\{\!\!\{\varphi_{init}(i,n,a)\}\!\!\}\ SeqInit\ \{\!\!\{\varphi_{error}(n,a)\}\!\!\}$$

where:
(i) $\varphi_{init}(i,n,a)$ is $i \geq 0 \ \wedge\ n = dim(a) \ \wedge\ n \geq 1$, and
(ii) $\varphi_{error}(n,a)$ is $\exists j\ (0 \leq j \ \wedge\ j+1 < n \ \wedge\ a[j] \geq a[j+1])$.

First, the above incorrectness triple is translated into a CLP(Array) program T. In particular, the properties φ_{init} and φ_{error} are defined by the following clauses, respectively:

```
1. phiInit(I,N,A) :- I≥0, dim(A,N), N≥1.
2. phiError(N,A) :- Z=W+1, W≥0, W+1<N, U≥V, read(A,W,U), read(A,Z,V).
```

The clauses defining the predicates `initConf` and `errorConf` which specify the initial and the error configurations, respectively, are as follows:

```
3. initConf(cf(cmd(l0,Cmd),Ps)):-at(l0,Cmd),progState(Ps),phiInit(Ps).
4. errorConf(cf(cmd(lh,Cmd),Ps)):-at(lh,Cmd),progState(Ps),phiError(Ps).
```

The predicates `at` and `progState` are defined by: '`at(l0, asgn(int(i), int(1))).`', '`at(lh, halt).`', and '`progState([[int(i), I], [int(n), N], [array(a), A]]).`'.

Now we apply Step (A) of our verification method, which consists in the removal of the interpreter. From program T we obtain the following program $T1$:

```
5. incorrect :- Z=W+1, W≥0, W+1<N, U≥V, N≤I,
                read(A,W,U), read(A,Z,V), p(I,N,A).
6. p(I1,N,B) :- 1≤I, I<N, D=I−1, I1=I+1, V=U+1,
                read(A,D,U), write(A,I,V,B), p(I,N,A).
7. p(I,N,A) :- I=1, N≥1.
```

The CLP(Array) program $T1$ expresses the verification conditions for *SeqInit*. Indeed, predicate `p` is an invariant for the `while` loop. For reasons of simplicity, the predicates expressing the assertions associated with assignments and conditionals have been unfolded away during the removal of the interpreter. (The strategy for removing the interpreter can be customized.)

Due to the presence of integer and array variables, the least $\mathcal{A}$-model $M(T1)$ may be infinite, and both the bottom-up and top-down evaluation of the goal `:- incorrect` may not terminate (indeed, this is the case in our example above). Thus, we cannot directly use the standard CLP systems to prove program correctness. In order to cope with this difficulty, we use a method based on CLP program transformations, which allows us to avoid the exhaustive exploration of the possibly infinite space of reachable configurations.

5 A Transformation Strategy for Verification

As mentioned above, the verification conditions expressed as the CLP(Array) program $T1$ generated by Step (A) are satisfiable iff `incorrect` $\notin M(T1)$. Our verification method is based on the fact that by transforming the CLP(Array) program $T1$ using rules that preserve the least $\mathcal{A}$-model, we get a new CLP(Array) program $T2$ that expresses equisatisfiable verification conditions.

Step (B) has the objective of showing, through further transformations, that *either* the verification conditions generated by Step (A) are satisfiable (that is, `incorrect`$\notin M(T1)$ and hence *prog* is correct with respect to φ_{init} and φ_{error}), *or* they are unsatisfiable (that is, `incorrect` $\in M(T1)$ and hence *prog* is not correct with respect to φ_{init} and φ_{error}). To this aim, Step (B) propagates the initial and/or the error properties so as to derive from program $T1$ a program $T2$ where the predicate `incorrect` is defined by either (α) the fact '`incorrect`' (in which case the verification conditions are unsatisfiable and *prog* is incorrect), or (β) the empty set of clauses (in which case the verification conditions are satisfiable and *prog* is correct). In the case where neither (α) nor (β) holds, that is, in program $T2$ the predicate `incorrect` is defined by a non-empty set of clauses not containing the fact '`incorrect`', we cannot conclude anything about the correctness of *prog*. However, similarly to what has been proposed in [8], in this case we can iterate Step (B), alternating the propagation of the initial and error properties, in the hope of deriving a program where either (α) or (β) holds. Obviously, due to undecidability limitations, it may be the case that we never get a program where either (α) or (β) holds.

Step (B) is performed by applying the unfold/fold transformation rules according to the *Transform* strategy shown in Figure 1. *Transform* can be viewed as a backward propagation of the error property. The forward propagation of the initial property can be obtained by combining *Transform* with the *Reversal* transformation described in [8]. For lack of space we do not present this extra transformation here.

```
Input: A linear CLP(Array) program $T1$.
Output: Program $T2$ such that incorrect $\in M(T1)$ iff incorrect $\in M(T2)$.

INITIALIZATION:
Let InDefs be the set of all clauses of $T1$ whose head is the atom incorrect;
$T2 := \emptyset$;   $Defs := InDefs$;

while in InDefs there is a clause $C$ do
    UNFOLDING: Unfold $C$ w.r.t. the unique atom in its body using $T1$, and derive
        a set $U(C)$ of clauses;
    CONSTRAINT REPLACEMENT: Apply a sequence of constraint replacements by
        using the Laws of Arrays, and derive from $U(C)$ a set $R(C)$ of clauses;
    CLAUSE REMOVAL: Remove from $R(C)$ all clauses whose body contains an un-
        satisfiable constraint;
    DEFINITION & FOLDING: Introduce a (possibly empty) set of new predicate def-
        initions and add them to Defs and to InDefs;
        Fold the clauses in $R(C)$ different from constrained facts by using the clauses
        in Defs, and derive a set $F(C)$ of clauses;
    $InDefs := InDefs - \{C\}$;   $T2 := T2 \cup F(C)$;
end-while;

REMOVAL OF USELESS CLAUSES:
Remove from $T2$ all clauses with head predicate p, if in $T2$ there is no constrained fact
q(...) :- c where q is either p or a predicate on which p depends.
```

Fig. 1. The *Transform* strategy

The input program $T1$ is a *linear* CLP(Array) program (we can show, in fact, that Step (A) always generates a linear program).
UNFOLDING performs one inference step backward from the error property.
The CONSTRAINT REPLACEMENT phase by applying the theory of arrays, infers new constraints on the variables of the only atom that occurs in the body of each clause obtained by the UNFOLDING phase. It works as follows. We select a clause, say `H :- c, G`, in the set $U(C)$ of the clauses obtained by unfolding, and

we replace that clause by the one(s) obtained by applying as long as possible the following rules. Note that this process always terminates and, in general, it is nondeterministic.

(RR1) If $\texttt{c} \sqsubseteq (\texttt{I}=\texttt{J})$ then
replace: `read(A,I,U), read(A,J,V)` by: `U=V, read(A,I,U)`

(RR2) If $\texttt{c} \equiv$ (`read(A,I,U),read(A,J,V),d`), $\texttt{d} \not\sqsubseteq (\texttt{I}\neq\texttt{J})$, and $\texttt{d} \sqsubseteq (\texttt{U}\neq\texttt{V})$ then
add to `c` the constraint: $\texttt{I}\neq\texttt{J}$

(WR1) If $\texttt{c} \sqsubseteq (\texttt{I}=\texttt{J})$ then
replace: `write(A,I,U,B), read(B,J,V)`
by: `U=V, write(A,I,U,B)`

(WR2) If $\texttt{c} \sqsubseteq (\texttt{I}\neq\texttt{J})$ then
replace: `write(A,I,U,B), read(B,J,V)`
by: `write(A,I,U,B), read(A,J,V)`

(WR3) If $\texttt{c} \not\sqsubseteq \texttt{I}=\texttt{J}$ and $\texttt{c} \not\sqsubseteq \texttt{I}\neq\texttt{J}$ then
replace: `H :- c, write(A,I,U,B), read(B,J,V), G`
by: `H :- c, I=J, U=V, write(A,I,U,B), G`
and `H :- c,` $\texttt{I}\neq\texttt{J}$`, write(A,I,U,B), read(A,J,V), G`

Rules RR1 and RR2 are derived from the array axiom A1 (see Section 3), and rules WR1–WR3 are derived from the array axioms A2 and A3 (see Section 3).

The DEFINITION & FOLDING phase introduces new predicate definitions by suitable generalizations of the constraints. These generalizations guarantee the termination of *Transform*, but at the same time they should be as specific as possible in order to achieve maximal precision. This phase works as follows. Let $C1$ in $R(C)$ be a clause of the form `H :- c, p(X)`. We assume that *Defs* is structured as a tree of clauses, where clause A is the parent of clause B if B has been introduced for folding a clause in $R(A)$. If in *Defs* there is (a variant of) a clause D: `newp(X) :- d, p(X)` such that $\mathit{vars}(\texttt{d}) \subseteq \mathit{vars}(\texttt{c})$ and $\texttt{c} \sqsubseteq \texttt{d}$, then we fold $C1$ using D. Otherwise, we introduce a clause of the form `newp(X) :- gen, p(X)` where: (i) `newp` is a predicate symbol occurring neither in the initial program nor in *Defs*, and (ii) `gen` is a constraint such that $\mathit{vars}(\texttt{gen}) \subseteq \mathit{vars}(\texttt{c})$ and $\texttt{c} \sqsubseteq \texttt{gen}$. The constraint `gen` is called a *generalization* of the constraint `c` and is constructed as follows.

Let `c` be of the form $\texttt{i}_1, \texttt{rw}_1$, where $\texttt{i}_1$ is an integer constraint and $\texttt{rw}_1$ is a conjunction of `read` and `write` constraints.

(1) Delete all `write` constraints from $\texttt{rw}_1$, hence deriving $\texttt{r}_1$.
(2) Rewrite $\texttt{i}_1, \texttt{r}_1$ so that all occurrences of integers in `read` constraints are distinct variables not appearing in `X` (this can be achieved by possibly adding some integer equalities to $\texttt{r}_1$), hence deriving $\texttt{i}_2, \texttt{r}_2$.
(3) Compute the projection $\texttt{i}_3$ (in the rationals) of the constraint $\texttt{i}_2$ onto $\mathit{vars}(\texttt{r}_2) \cup \{\texttt{X}\}$ (thus $\texttt{i}_2 \sqsubseteq \texttt{i}_3$ in the domain of the integers).

(4) Delete from $\texttt{r}_2$ all `read(A, I, V)` constraints such that either (i) `A` does not occur in `X` or (ii) `V` does not occur in $\texttt{i}_3$, thereby deriving a new value for $\texttt{r}_2$. If at least one `read` has been deleted from $\texttt{r}_2$ then go to step (3).

Let $\texttt{i}_3, \texttt{r}_3$ be the constraint obtained after the applications of steps (3)–(4).

(5) *If* in *Defs* there is an ancestor (defined as the reflexive, transitive closure of the parent relation) of C of the form $\texttt{H}_0$:- $\texttt{i}_0, \texttt{r}_0, \texttt{p(X)}$ such that $\texttt{r}_0, \texttt{p(X)}$ is a subconjunction of $\texttt{r}_3, \texttt{p(X)}$,

Then compute a generalization `g` of the constraints $\texttt{i}_3$ and $\texttt{i}_0$ such that $\texttt{i}_3 \sqsubseteq \texttt{g}$, by using a *generalization operator* for linear constraints (we refer to [7,14,33] for generalization operators based on *widening*, *convex hull*, and *well-quasi orderings*). Define the constraint `gen` as $\texttt{g}, \texttt{r}_0$;

Else define the constraint `gen` as $\texttt{i}_3, \texttt{r}_3$.

The correctness of the strategy with respect to the least $\mathcal{A}$-model semantics follows from Theorem 1, by observing that every clause defining a new predicate introduced by Definition & Folding is unfolded once during the execution of the strategy (indeed every such clause is added to *InDefs*).

The termination of the *Transform* strategy is based on the following facts:
(i) Constraint satisfiability and entailment are checked by a terminating solver (note that completeness is not necessary for the termination of *Transform*).
(ii) Constraint Replacement terminates (see above).
(iii) The set of new clauses that, during the execution of the *Transform* strategy, can be introduced by Definition & Folding steps is finite. Indeed, by construction, they are all of the form `H :- i, r, p(X)`, where: (1) `X` is a tuple of variables, (2) `i` is an integer constraint, (3) `r` is a conjunction of array constraints of the form `read(A, I, V)`, where `A` is a variable in `X` and the variables `I` and `V` occur in `i` only, (4) the cardinality of `r` is bounded, because generalization does not introduce a clause `newp(X)` :- $\texttt{i}_3, \texttt{r}_3, \texttt{p(X)}$ if there exists an ancestor clause of the form $\texttt{H}_0$:- $\texttt{i}_0, \texttt{r}_0, \texttt{p(X)}$ such that $\texttt{r}_0, \texttt{p(X)}$ is a subconjunction of $\texttt{r}_3, \texttt{p(X)}$, (5) we assume that the generalization operator on integer constraints has the following *finiteness* property: only finite chains of generalizations of any given integer constraint can be generated by applying the operator. The already mentioned generalization operators presented in [7,14,33] satisfy this finiteness property.

Theorem 2. (Termination and Correctness of the *Transform* strategy) (i) *The Transform strategy terminates.* (ii) *Let program $T2$ be the output of the Transform strategy applied on the input program $T1$. Then,* `incorrect` $\in M(T1)$ *iff* `incorrect` $\in M(T2)$.

Let us now consider again the *SeqInit* example of Section 4 and perform Step (B). We apply the *Transform* strategy starting from program $T1$.

Unfolding. First, we unfold clause 5 w.r.t. the atom `p(I, N, A)`, and we get:

8. `incorrect :- Z=W+1, W≥0, Z≤I, D=I−1, N=I+1, Y=X+1, U≥V,`
   `read(B, W, U), read(B, Z, V), read(A, D, X), write(A, I, Y, B), p(I, N, A).`

Constraint Replacement. Then, by applying the replacement rules WR2, WR3, and RR1 to clause 8, we get the following clause:

9. `incorrect :- Z=W+1, W≥0, Z<I, D=I−1, N=I+1, Y=X+1, U≥V,`
`read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), p(I,N,A).`

In particular, since W≠I is entailed by the constraint in clause 8, we apply rule WR2 and we obtain a new clause, say 8.1, where read(B,W,U), write(A,I,Y,B) is replaced by read(A,W,U), write(A,I,Y,B). Then, since neither Z=I nor Z≠I is entailed by the constraint in clause 8.1, we apply rule WR3 and we obtain two clauses 8.2 and 8.3, where the constraint read(B,Z,V), write(A,I,Y,B) is replaced by Z = I, Y = V, write(A,I,Y,B) and Z ≠ I, read(A,Z,U), write(A,I,Y,B), respectively. Finally, since D = W is entailed by the constraint in clause 8.2, we apply rule RR1 to clause 8.2 and we add the constraint X = U to its body, hence deriving the unsatisfiable constraint X = U, Y = X + 1, Y = V, U ≥ V. Thus, the clause derived by the latter replacement is removed. Clause 9 is derived from 8.3 by rewriting Z ≤ I, Z ≠ I as Z < I.

Definition & Folding. In order to fold clause 9 we introduce a new definition by applying Steps (1)–(5) of the Definition & Folding phase. In particular, by deleting the write constraint (Step 1) and projecting the integer constraint (Step 3), we get a constraint where the variable X occurs in read(A,D,X) only. Thus, we delete read(A,D,X) (Step 4). Finally, we compute a generalization of the constraints occurring in clauses 5 and 9 by using the convex hull operator (Step 5). We get:

10. `new1(I,N,A) :- Z=W+1, W≥0, N≤I+1, N≥W+2, W≤I−2, U≥V,`
`read(A,W,U), read(A,Z,V), p(I,N,A).`

By folding clause 9 using clause 10, we get:

11. `incorrect :- Z=W+1, W≥0, Z<I, D=I−1, N=I+1, Y=X+1, U≥V,`
`read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), new1(I,N,A).`

Now we proceed by performing a second iteration of the body of the while-loop of the *Transform* strategy because *InDefs* is not empty (indeed, at this point clause 10 belongs to *InDefs*).

Unfolding. After unfolding clause 10 we get the following clause:

12. `new1(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1,`
`Z≤I, Z≥1, N>I, U≥V, read(B,W,U), read(B,Z,V),`
`read(A,D,X), write(A,I,Y,B), p(I,N,A).`

Constraint Replacement. Then, by applying rules RR1, WR2, and WR3 to clause 12, we get the following clause:

13. `new1(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1,`
`Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V),`
`read(A,D,X), write(A,I,Y,B), p(I,N,A).`

Definition & Folding. In order to fold clause 13 we introduce the following clause, whose body is derived by computing the widening [5,7] of the integer constraints in the ancestor clause 10 with respect to the integer constraints in clause 13:

```
14. new2(I,N,A) :- Z=W+1, W≥0, W≤I−1, N>Z, U≥V,
                   read(A,W,U), read(A,Z,V), p(I,N,A).
```

By folding clause 13 using clause 14, we get:

```
15. new1(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1,
             Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V),
             read(A,D,X), write(A,I,Y,B), new2(I,N,A).
```

Now we perform the third iteration of the body of the while-loop of the strategy starting from the newly introduced definition, that is, clause 14. After some unfolding and constraint replacement steps, followed by a final folding step, from clause 14 we get:

```
16. new2(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, I≥1,
             Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V),
             read(A,D,X), write(A,I,Y,B), new2(I,N,A).
```

The final transformed program is made out of clauses 11, 15, and 16. Since this program has no constrained facts, by the last step of the *Transform* procedure we derive the empty program $T2$, and we conclude that the program *SeqInit* is correct with respect to the given φ_{init} and φ_{error} properties.

6 Experimental Evaluation

We have performed an experimental evaluation of our method on a benchmark set of programs acting on arrays, mostly taken from the literature [3,12,21,27]. The results of our experiments, which are summarized in Tables 1 and 2, show that our approach is effective and quite efficient in practice.

Our verifier consists of a module, based on the C Intermediate Language (CIL) [32], which translates a C program together with the initial and error configurations, into a set of CLP(Array) facts, and a module for CLP(Array) program transformation that removes the interpreter and applies the *Transform* strategy. The latter module is implemented using the MAP system [29], a tool for transforming constraint logic programs written in SICStus Prolog.

We now briefly discuss the programs we have used for our experimental evaluation (see Table 1 where we have also indicated the properties we have verified).

Some programs deal with array initialization: program *init* initializes all the elements of the array to a constant, while *init-non-constant* and *init-sequence* use expressions which depend on the element position and on the preceding element, respectively. Program *init-partial* initializes only an initial portion of the array. Program *copy* performs the element-wise copy of an entire array to another array, while *copy-partial* copies only an initial portion of the array, and the program *copy-reverse* copies the array in reverse order. The program *max* computes the maximum of an array. The programs *sum* and *difference* perform the element-wise sum and difference, respectively, of two input arrays.

Table 1. Benchmark array programs. Variables `a,b,c` are arrays of integers of size `n`.

Program	*Code*	*Verified Property*
init	`for(i=0; i<n; i++) a[i]=c;`	$\forall i.\ (0 \leq i \wedge i < n) \rightarrow a[i] = c$
init-partial	`for(i=0; i<k; i++) a[i]=0;`	$\forall i.\ (0 \leq i \wedge i < k \wedge k \leq n) \rightarrow a[i] = 0$
init-non-constant	`for(i=0; i<n; i++) a[i]=2*i+c;`	$\forall i.\ (0 \leq i \wedge i < n) \rightarrow a[i] = 2*i+c$
init-sequence	`a[0]=7; i=1; while(i<n) { a[i]=a[i-1]+1; i++;}`	$\forall i.\ (1 \leq i \wedge i < n) \rightarrow a[i] = a[i-1]+1$
copy	`for(i=0; i<n; i++) a[i]=b[i];`	$\forall i.\ (0 \leq i \wedge i < n) \rightarrow a[i] = b[i]$
copy-partial	`for(i=0; i<k; i++) a[i]=b[i];`	$\forall i.\ (0 \leq i \wedge i < k \wedge k \leq n) \rightarrow a[i] = b[i]$
copy-reverse	`for(i=0; i<n; i++) b[i]=a[i]; for(i=0; i<n; i++) a[i]=b[n-i-1];`	$\forall i.\ (0 \leq i \wedge i < n) \rightarrow a[i] = b[n-i-1]$
max	`m=a[0]; i=1; while(i<n) { if(a[i]>m) m=a[i]; i++; }`	$\forall i.\ (0 \leq i \wedge i < n \wedge n \geq 1) \rightarrow m \geq a[i]$
sum	`for(i=0; i<n; i++) c[i]=a[i]+b[i];`	$\forall i.\ (0 \leq i \wedge i < n) \rightarrow c[i] = a[i]+b[i]$
difference	`for(i=0; i<n; i++) c[i]=a[i]-b[i];`	$\forall i.\ (0 \leq i \wedge i < n) \rightarrow c[i] = a[i]-b[i]$
find	`p=-1; for(i=0; i<n; i++) if(a[i]==e) { p=i; break; }`	$(0 \leq p \wedge p < n) \rightarrow a[p] = e$
first-not-null	`s=n; for(i=0; i<n; ++i) if(s==n && a[i]!=0) s=i;`	$(0 \leq s \wedge s < n) \rightarrow (\ a[s] \neq 0 \wedge (\forall i.\ (0 \leq i \wedge i < s) \rightarrow a[i] = 0)\ )$
find-first-non-null	`p=-1; for(i=0; i<n; i++) if(a[i]!=0) { p=i; break; }`	$(0 \leq p \wedge p < n) \rightarrow a[p] \neq 0$
partition	`i=0; j=0; k=0; while(i<n) { if(a[i]>=0) { b[j]=a[i]; j++; } else { c[k]=a[i]; k++; } ++i; }`	$(\forall i.\ (0 \leq i \wedge i < j) \rightarrow b[i] \geq 0) \wedge (\forall i.\ (0 \leq i \wedge i < k) \rightarrow c[i] < 0)$
insertionsort-inner	`x=a[i]; j=i-1; while(j>=0 && a[j]>x) { a[j+1]=a[j]; --j; }`	$\forall k.\ (0 \leq i \wedge i < n \wedge j{+}1 < k \wedge k \leq i) \rightarrow a[k] > x$
bubblesort-inner	`for(j=0; j<n-i-1; j++) { if(a[j] > a[j+1]) { tmp = a[j]; a[j] = a[j+1]; a[j+1] = tmp; } }`	$\forall k.\ (0 \leq i \wedge i < n \wedge 0 \leq k \wedge k < j \wedge j = n-i-1) \rightarrow a[k] \leq a[j]$
selectionsort-inner	`for(j=i+1; j<n; j++) { if(a[i]>a[j]) { tmp=a[i]; a[i]=a[j]; a[j]=tmp; } }`	$\forall k.(0 \leq i \wedge i \leq k \wedge k < n) \rightarrow a[k] \geq a[i]$

The program *find* looks for a particular value inside an array and returns the position of its first occurrence, if any, or a negative value otherwise. The programs *find-first-non-null* and *first-not-null* are two programs which return the position of the first non-zero element. For these programs, differently from [12,21], we prove that when the search succeeds, the returned position contains a non-zero element and we also proved that all the preceding elements are zero elements. The program *partition* copies non-negative and negative elements of the array into two distinct arrays. The programs *insertionsort-inner*, *bubblesort-inner*, and *selectionsort-inner* are based on textbook implementations of sorting algorithms. The source code of all the verification problems we have considered is available at `http://map.uniroma2.it/smc/`.

For verifying the above programs we have applied the *Transform* strategy using different generalization operators, which are based on the widening and convex hull operators. In particular the Gen_W and Gen_S operators use the *Widen* and *CHWidenSum* operators between constraints [14].

We have also combined these operators with a delay mechanism which, before starting the actual generalization process, introduces a definition which is computed by using convex hull alone, without widening. We denote by Gen_{WD} and Gen_{SD} the operators obtained by combining delayed generalization with the *Widen* and *CHWidenSum* operators, respectively.

In Table 2 we report the results obtained by applying *Transform* with the four generalization operators mentioned above. The first column contains references to papers where the program verification example has been considered.

The last four columns are labeled with the name of the generalization operator. For each program proved correct we report the time in seconds taken to verify the property of interest. By *unknown* we indicate that *Transform* derives a CLP(Array) program containing constrained facts different from '`incorrect`', and hence the satisfiability (or the unsatisfiability) of the corresponding verification conditions cannot be checked.

We also report, for each generalization operator, the number of successfully verified programs (which measures the *precision* of the operator), the *total time* taken to run the whole benchmark and the *average time* per successful answer, respectively.

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

The data presented in Table 2 show that by using the Gen_W operator, which applies the widening operator alone, our method is only able to prove 7 programs out of 17. However, precision can be recovered by applying the convex hull operator when introducing new definitions, possibly combined with widening.

The best trade-off between precision and performance is provided by the Gen_{WD} operator which is able to prove all 17 programs with an average time of 0.92 *s*. In this case the use of the delay mechanism, which uses convex hull, suffices to compensate the weakness demonstrated by the use of widening alone. Note also that one program, *init-sequence*, can only be proved by applying operators which use delayed generalization. This confirms the effectiveness of the convex

Table 2. Verification results using the MAP system with different generalization operators. Times are in seconds.

Program	*References*	Gen_W	Gen_{WD}	Gen_S	Gen_{SD}
init	[3,12,37]	*unknown*	0.06	0.10	0.08
init-partial	[3,12]	*unknown*	0.06	0.07	0.08
init-non-constant	[3,12,27,37]	*unknown*	0.06	0.22	0.22
init-sequence	[21,27]	*unknown*	0.80	*unknown*	1.20
copy	[3,12,21,27,37]	*unknown*	0.27	0.33	0.29
copy-partial	[3,12]	*unknown*	0.29	0.34	0.34
copy-reverse	[3,12]	*unknown*	0.27	0.46	0.45
max	[21,27]	*unknown*	0.31	0.24	0.33
sum		*unknown*	0.68	1.14	1.12
difference	[3]	*unknown*	0.66	1.15	1.11
find	[3,12]	0.25	0.43	0.46	0.45
first-not-null	[21]	0.38	0.41	0.42	0.42
find-first-non-null	[3,12]	1.24	1.87	1.94	1.93
partition	[12,27,37]	0.06	0.11	0.14	0.12
insertionsort-inner	[21,27,37]	0.21	0.26	0.45	0.43
bubblesort-inner		2.46	2.71	2.45	2.75
selectionsort-inner	[37]	7.20	6.40	7.23	7.16
	precision	7	17	16	17
	total time	11.80	15.65	17.14	18.48
	average time	1.69	0.92	1.07	1.09

hull operator which may help inferring relations among program variables, and may ease the discovery of useful program invariants, while determining (in our set of examples) only a slight increase of verification times.

A detailed comparison of the performance of our system with respect to the other verification systems referred to in Table 1 is difficult to make at this time because the systems are not all readily available and also the results reported in the literature do not refer to the same code for the input C programs.

7 Related Work and Conclusions

The verification method presented in this paper is an extension of the one introduced in [8], where programs manipulating arrays were not considered. Some examples suggesting how arrays and recursively defined properties can be dealt with in our transformational approach were presented in [9], where, however, no automatic strategy was presented. In this paper we have shown that by applying a quite simple and general automated transformation strategy it is possible to prove most of the examples found in the literature, with reasonable performance. We are currently extending our strategy to deal with recursive programs, such as *quicksort*.

The idea of encoding imperative programs into CLP programs for reasoning about their properties was presented in various papers [15,23,34], which show that through CLP programs one can express in a simple manner both (i) the symbolic executions of imperative programs, and (ii) the invariants that hold during their executions. The peculiarity of our work with respect to [15,23,34] is that we use CLP program transformations to prove properties, instead of (symbolic) execution or static analysis.

The verification method presented in this paper is also related to several other methods that use abstract interpretation and theorem proving techniques.

Now we briefly report on related papers which use abstract interpretations for finding invariants of programs that manipulate arrays. In [21], which builds upon [18], invariants are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. A similar approach is followed in [6] where a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs is introduced. In [16,26] a predicate abstraction for inferring universally quantified properties of array elements is presented, and in [20] the authors present a similar technique which uses template-based quantified abstract domains.

Methods based on abstract interpretation construct overapproximations, that is, invariants implied by the program executions. This approach has the advantage of being quite efficient because it fixes in advance a finite set of assertions where the invariants are searched for, but for the same reason it may lack flexibility as the abstraction should be re-designed when the verification fails.

Also theorem provers have been used for discovering invariants in programs which manipulate arrays and prove verification conditions generated from the programs. In particular, in [4] a satisfiability decision procedure for a decidable fragment of a theory of arrays is presented. That fragment is expressive enough to prove properties such as sortedness of arrays. In [24,25,31] the authors present some techniques based on theorem proving which may generate array invariants. In [37] a backward reachability analysis based on predicate abstraction and abstraction refinement is used for verifying assertions which are universally quantified over array indexes. Finally, we would like to mention that techniques based on Satisfiability Modulo Theory (SMT) have been applied for generating and verifying universally quantified properties over array variables (see, for instance, [1,27]).

The approaches based on theorem proving and SMT are more flexible with respect to those based on abstract interpretation because no finite set of abstractions is fixed in advance, but the suitable assertions needed by the proof are generated on the fly.

Although the approach based on CLP program transformation shares many ideas and techniques with abstract interpretation and automated theorem proving, we believe that it has some distinctive features that make it quite appealing. Indeed, this paper and previous work (such as [8,14,34]) show that one can construct a framework where the generation of verification conditions and their verification can both be viewed as program transformations. The approach is parametric with respect to the program syntax and semantics, because interpreters and proof systems can easily be written in CLP, and verification conditions can automatically be generated by specialization. Moreover, optimizing transformations can be applied to improve the efficiency of verification. Finally, transformations can easily be composed together to derive very sophisticated verification techniques. For instance, in [8] it is shown that the *iteration* of specialization combined with the reversal of the direction used for constraint propagation can significantly improve the precision of verification.

In order to further validate our approach, we plan to address the issue of proving correctness of programs manipulating *dynamic data structures* such as lists or heaps, looking for a set of suitable constraint replacement laws which axiomatize those structures. For some specific theories we could also apply the constraint replacement rule by exploiting the results obtained by external theorem provers or Satisfiability Modulo Theory solvers.

An interesting direction for future research is also the combination of transformations that guarantee equisatisfiability of verification conditions (like the ones considered in this paper) together with other techniques for checking the satisfiability of constrained Horn clauses.

Acknowledgements. We would like to thank the anonymous referees for their helpful comments and constructive criticism.

References

1. 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)
2. Bjørner, N., McMillan, K., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT 2012, pp. 3–11 (2012)
3. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM (1977)
6. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL 2011, pp. 105–118 (2011)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, pp. 84–96. ACM (1978)
8. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying Programs via Iterated Specialization. In: PEPM 2013, pp. 43–52. ACM (2013)
9. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verification of Imperative Programs by Constraint Logic Program Transformation. In: SAIRP 2013, Festschrift for Dave Schmidt. Electronic Proceedings in Theoretical Computer Science, vol. 129, pp. 186–210 (2013)
10. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)

11. 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)
12. 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)
13. Etalle, S., Gabbrielli, M.: Transformations of CLP modules. Theoretical Computer Science 166, 101–146 (1996)
14. Fioravanti, F., Pettorossi, A., Proietti, M., Senni, V.: Generalization strategies for the verification of infinite state systems. Theory and Practice of Logic Programming 13(2), 175–199 (2013)
15. Flanagan, C.: Automatic software model checking via constraint logic. Sci. Comput. Program. 50(1-3), 253–270 (2004)
16. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL 2002, pp. 191–202. ACM, New York (2002)
17. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. Ann. Math. Artif. Intell. 50(3-4), 231–254 (2007)
18. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: POPL 2005, pp. 338–350. ACM (2005)
19. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A Software Verifier based on Horn Clauses. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
20. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
21. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI 2008, pp. 339–348 (2008)
22. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19/20, 503–581 (1994)
23. Jaffar, J., Santosa, A.E., Voicu, R.: An interpolation method for CLP traversal. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 454–469. Springer, Heidelberg (2009)
24. 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)
25. 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)
26. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Log. 9(1) (2007)
27. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 169–188. Springer, Heidelberg (2013)
28. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
29. The MAP transformation system, `http://www.iasi.cnr.it/~proietti/system.html`
30. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing: Proc. of IFIP 1962, pp. 21–28. North Holland, Amsterdam (1963)
31. 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)

32. 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. 209–265. Springer, Heidelberg (2002)
33. Peralta, J.C., Gallagher, J.P.: Convex hull abstractions in specialization of CLP programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 90–108. Springer, Heidelberg (2003)
34. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 246–261. Springer, Heidelberg (1998)
35. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
36. Reynolds, C.J.: Theories of Programming Languages. Cambridge Univ. Press (1998)
37. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
38. van Emden, M.H., Kowalski, R.: The semantics of predicate logic as a programming language. Journal of the ACM 23(4), 733–742 (1976)

Weakest Precondition Synthesis for Compiler Optimizations

Nuno P. Lopes and José Monteiro

INESC-ID, IST Universidade de Lisboa

Abstract. Compiler optimizations play an increasingly important role in code generation. This is especially true with the advent of resource-limited mobile devices. We rely on compiler optimizations to improve performance, reduce code size, and reduce power consumption of our programs.

Despite being a mature field, compiler optimizations are still designed and implemented by hand, and usually without providing any guarantee of correctness.

In addition to devising the code transformations, designers and implementers have to come up with an analysis that determines in which cases the optimization can be safely applied. In other words, the optimization designer has to specify a precondition that ensures that the optimization is semantics-preserving. However, devising preconditions for optimizations by hand is a non-trivial task. It is easy to specify a precondition that, although correct, is too restrictive, and therefore misses some optimization opportunities.

In this paper, we propose, to the best of our knowledge, the first algorithm for the automatic synthesis of preconditions for compiler optimizations. The synthesized preconditions are provably correct by construction, and they are guaranteed to be the weakest in the precondition language that we consider.

We implemented the proposed technique in a tool named PSyCO. We present examples of preconditions synthesized by PSyCO, as well as the results of running PSyCO on a set of optimizations.

1 Introduction

Compiler optimizations are increasingly important. We rely on them to improve performance, reduce code size, and reduce power consumption of our programs. The advent of mobile devices with limited resources and the need to reduce the operating costs of data centers puts even more pressure on the quality of the results of compiler optimizations.

This demand for improving code efficiency is driving the development of new and more complex optimizations. However, neither the specification nor the implementation of these optimizations is usually proved correct.

In fact, a recent study found bugs in all the most used compilers [39]. These bugs range from mere crashes to subtle wrong-code emission.

Ensuring that compilers are correct is of extreme importance. All the programs we produce, in one way or another, are processed by compilers. If the compilers are not proved correct, properties formally verified at the source-code level of a program are not carried to the binary code, since the compiler may introduce bugs during the translation process.

Despite such a strong necessity for compilation quality, compilers are still largely written by hand. Moreover, the source-code of each of the major compilers has several million lines of code. Testing a whole code base is therefore unlikely to be practical to accomplish.

This paper gives a step towards improving the situation. We present, to the best of our knowledge, the first algorithm for the automatic synthesis of the *weakest* precondition for compiler optimizations specified in a high-level language. These preconditions are provably correct by construction.

We consider a language of preconditions for compiler optimizations consisting of read and write sets for template statements (and expressions), which we believe to be adequate to express the most used conditions in this domain. Given a compiler optimization specified in a high-level template language, our algorithm synthesizes the *weakest* precondition in terms of read and write sets (assuming it is solely expressible in terms of these sets).

The generated preconditions can then be either used by the compiler developer to implement an analysis that guarantees that the precondition holds before performing the code transformation, or it can be used by a separate tool to automatically generate such an analysis.

The algorithm works in a counterexample-driven way. It requires a black box that can prove the correctness of a compiler optimization, or produce a counterexample otherwise. Then, the algorithm processes the counterexample and produces the weakest precondition guaranteed to prevent that counterexample. The algorithm repeats this process until no counterexamples are possible, i.e., when the optimization is correct (which is guaranteed to occur, since the set of possible preconditions is finite).

Our algorithm is more generally applicable than the domain of compiler optimizations. The algorithm can synthesize weakest preconditions for any problem where the set of preconditions is finite, although possibly too large to test each case individually, as long as there exists an oracle that can prove correctness or produce counterexamples if not correct.

The rest of the paper is organized as follows. Section 2 gives a set of preliminary definitions. Section 3 gives an intuition of how our algorithm synthesizes weakest preconditions for compiler optimizations through a simple example. Section 4 describes our algorithm for the synthesis of weakest preconditions for compiler optimizations. Section 5 presents PSyCO, a tool that implements the proposed algorithm, as well as examples of preconditions generated by PSyCO and results of running it over a set of compiler optimizations. Section 6 presents the related work.

$$
\begin{aligned}
e &::= n \mid v \mid e_1 \oplus e_2 \mid \mathsf{E}_i \\
b &::= e \leq 0 \mid b_1 \otimes b_2 \mid \neg b_1 \mid \mathsf{B}_i \\
c &::= \textbf{skip} \mid v \ := \ e \mid c_1 \ ; \ c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c_1 \mid \mathsf{S}_i
\end{aligned}
$$

Fig. 1. Template program syntax. n is an integer number; v is a variable name; E_i are integer template expressions (side-effect free); B_i are boolean template expressions (side-effect free); S_i are template statements; $\oplus$ is a binary operator over integer expressions (e.g., $+$, $-$); and $\otimes$ is a binary operator over boolean expressions (e.g., $\wedge$, $\vee$).

2 Preliminaries

Compiler optimizations are represented by a triple (τ, ψ, h). Transformation function $\tau \equiv Src \Rightarrow Tgt$ is a function that takes an instantiation of the source template program Src and returns the target template program Tgt properly instantiated. An instantiation of a template program is a mapping from all the template statements and expressions to concrete (without templates) statements/expressions. The precondition ψ is a sufficient condition that makes τ semantics-preserving. Finally, h is a profitability heuristic, which states under which conditions the compiler should apply τ, since τ may not always be performance improving. We will ignore the profitability heuristic for the rest of this paper because it does not interfere with the correctness of optimizations.

Template Programs. Template programs are specified using the syntax shown in Figure 1. In addition to the normal program features, template programs may have template expressions and template statements. Template expressions are side-effect free expressions whose value is unknown. It may be a constant or it may be an arbitrary algebraic expression that depends on several variables. Similarly, template statements are placeholders for arbitrary statements (e.g., variable assignments, function calls, or even loops).

Side-effect free expressions are allowed to read any number of variables and memory locations (including none), but are not allowed to write to them. Moreover, side-effect free expressions may not raise exceptions nor trap. Such erroneous behaviors must be explicitly modeled in the control flow and/or as assignments to control variables.

Transformation functions state how each template statement/expression from the source program is transformed (e.g., moved, duplicated, eliminated) to produce the target program. For example, consider the following transformation function τ_1:

$$
\begin{array}{lcl}
\mathsf{S} & & v := \mathsf{E} \\
v := \mathsf{E} & \Rightarrow & \mathsf{S}
\end{array}
$$

As an example, we apply the transformation function τ_1 to the following program fragment:

$x := 0$
$v := x + 1$

The output of the transformation function is (with the instantiation $\mathsf{S} \mapsto x := 0,\ \mathsf{E} \mapsto x + 1$):

$v := x + 1$
$x := 0$

The transformed program is not equivalent to the original one, since if the initial value of x is not zero, then the programs will yield different values for v. Therefore, a necessary (but not sufficient) precondition to ensure that the transformation function is always semantics-preserving is that S cannot write to a variable that is read by E.

Preconditions of optimizations are specified as read and write sets of the template statements/expressions, which contain the variables that the template statements/expressions *may* read and write, respectively. Since template expressions are side-effect free, their write set is empty, i.e., for every template expressions E_i and B_i, we have $\mathsf{W}(\mathsf{E}_i) = \emptyset$ and $\mathsf{W}(\mathsf{B}_i) = \emptyset$.

For the previous example, we could use the condition $\mathsf{W}(\mathsf{S}) \cap \mathsf{R}(\mathsf{E}) = \emptyset \wedge v \notin \mathsf{R}(\mathsf{S}) \wedge v \notin \mathsf{W}(\mathsf{S})$ as the precondition. This precondition is sufficient to ensure that the transformation function is always semantics-preserving, and it would therefore rule out the instantiation above.

Let $\mathsf{Tmpl}(\tau)$ be the set of template statements/expressions of the transformation function τ. Let $\mathsf{Stmts}(\tau) \subseteq \mathsf{Tmpl}(\tau)$ be the set of template statements of the transformation function τ. In our example, we have $\mathsf{Tmpl}(\tau_1) = \{\mathsf{S}, \mathsf{E}\}$ and $\mathsf{Stmts}(\tau_1) = \{\mathsf{S}\}$.

Program Paths. A program path π is a sequence of straight-line statements (no loops nor **if** statements) and boolean expressions. For example, the path $i := 0\ ;\ i < n\ ;\ i := i + 1\ ;\ i \geq n$ could be a 1-step unrolling of a simple counting loop. We naturally extend $\mathsf{Stmts}(\pi)$ and $\mathsf{Tmpl}(\pi)$ to program paths.

Context Variables. Let $c_i \in C$ be a context variable. These variables c_i represent the variables that are possibly in scope where a program template may be instantiated (possibly none) and that do not appear in the transformation function.

For the example above we need two context variables and therefore we have $C = \{c_1, c_2\}$. Variable c_1 represents, e.g., the effects of S on x. While variable x does not appear explicitly in the transformation function, S does indeed modify x in the example instantiation. Variable c_2 represents all the other eventual variables of the program that are in scope (possibly none) that can be read by E and that cannot be written by S.

```
                              if B then
while I < N do                  while I < N do
  if B then                       S1
    S1                            I := I + 1
  else                  ⇒     else
    S2                          while I < N do
  I := I + 1                      S2
                                  I := I + 1
```

Fig. 2. Loop unswitching: the source template is on the left, and the target template is on the right

For every template statement S_i, we have that the context variable $c_i = \mathsf{CtxVar}(\mathsf{S}_i)$ is always in its write set, i.e., $c_i \in \mathsf{W}(\mathsf{S}_i)$. Therefore, each template statement may write to at least one distinct context variable.

In order to not restrict the generated preconditions, we require our precondition synthesis algorithm to be run with at least one more context variable than template statements, i.e., for transformation function τ we must have $|C| \geq |\mathsf{Stmts}(\tau)| + 1$. This lower bound on the size of C is sufficient to express all combinations of constraints of the form $\mathsf{R}(t) \cap \mathsf{W}(s_1) = \emptyset$ and $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) = \emptyset$ (and their respective negations) for any template t and statements s_1 and s_2. Constraints of the form $\mathsf{R}(t_1) \cap \mathsf{R}(t_2) = \emptyset$ are not considered, since we are not aware of any optimization requiring such kind of preconditions.

Let $\mathsf{Vars}(\tau)$ and $\mathsf{Vars}(\pi)$ be the set of variables in a transformation function τ or in a path π, respectively. Moreover, context variables are contained in these sets, i.e., $C \subseteq \mathsf{Vars}(\tau)$. In our example, we have $\mathsf{Vars}(\tau_1) = \{v, c_1, c_2\}$.

Miscellaneous. Throughout the paper, we use the constraint $x = \text{ite}(a, b, c)$ as the usual shorthand for $a \rightarrow x = b \wedge \neg a \rightarrow x = c$.

3 Illustrative Example

We illustrate our algorithm to synthesize weakest preconditions for compiler optimizations on a simple example. Figure 2 shows an optimization known as loop unswitching. Intuitively, this optimization looks correct iff B evaluates to the same boolean value in every iteration (i.e., B must be loop invariant).

S_1 and S_2 are template statements. They are placeholders that represent an arbitrary statement, such as a variable assignment, a loop, or a compound statement. Similarly, B is a template (side-effect free) boolean expression. We do not know what these statements and expressions exactly do (this is only defined when the optimization is applied to a specific piece of code), and so we need to derive a generic precondition to restrict their operation to guarantee that the transformation will be always semantics-preserving.

The language of preconditions we use for compiler optimizations is that of read and write sets of template statements/expressions. For example, to state that the template expression B cannot read variable I we use the notation $I \notin \mathsf{R}(\mathsf{B})$. This condition is actually necessary (but not sufficient) for the precondition of loop unswitching.

Our algorithm works in a counterexample-driven way. We require the existence of a black box that can prove the correctness of compiler optimizations, or produce a counterexample if the optimization is not correct.

For our example, starting with the precondition $P = \mathsf{true}$, we could get the following paths π_1 and π_2 (respectively, for the source and target templates):

$$I < N \; ; \; \mathsf{B} \; ; \; \mathsf{S}_1 \; ; \; I := I + 1 \; ; \; I < N \; ; \; \neg\mathsf{B} \; ; \; \mathsf{S}_2 \; ; \; I := I + 1 \; ; \; I \geq N$$

and:

$$\mathsf{B} \; ; \; I < N \; ; \; \mathsf{S}_1 \; ; \; I := I + 1 \; ; \; I < N \; ; \; \mathsf{S}_1 \; ; \; I := I + 1 \; ; \; I \geq N$$

Without any knowledge about S_1 and S_2, these paths are clearly a counterexample since S_1 and S_2 execute a different number of times in the source and target templates. For example, the instantiation $\mathsf{S}_1 \mapsto I := I + 1$, $\mathsf{S}_2 \mapsto I := I + 2$, $\mathsf{B} \mapsto I \leq 0$ makes the paths of the source and target programs terminate with different values for variable I. Therefore, we need to constrain the set of possible instantiations of S_1 and S_2 with a suitable precondition.

To generate a precondition for a given counterexample, we first encode the counterexample paths into logic in the usual way, with the variables of the target template being renamed in order to be different from the variables used in the source template. We explain only how template statements and expressions are encoded. Each template statement is treated as a conditional assignment to all variables of the program by a fresh variable. Then, for each pair of the same template symbol, we assert that their corresponding fresh variables are equal iff the values of their corresponding input variables that are in the read set are equal. Similarly, template expressions are replaced by fresh variables.

For our counterexample, we obtain the following constraint ϕ_1 for the source path (with $\mathsf{Vars}(\pi_1; \pi_2) = \{I, N, c_1, c_2, c_3\}$):

$$\begin{aligned}
&I_0 < N_0 \wedge \\
&B_0 \wedge \\
&I_1 = \text{ite}(w^I_{S1}, S1^I_0, I_0) \wedge N_1 = \text{ite}(w^N_{S1}, S1^N_0, N_0) \wedge \\
&c1_1 = \text{ite}(w^{c1}_{S1}, S1^{c1}_0, c1_0) \wedge c2_1 = \text{ite}(w^{c2}_{S1}, S1^{c2}_0, c2_0) \wedge c3_1 = \text{ite}(w^{c3}_{S1}, S1^{c3}_0, c3_0) \\
&\wedge I_2 = I_1 + 1 \wedge \\
&I_2 < N_1 \wedge \\
&\neg B_1 \wedge \\
&I_3 = \text{ite}(w^I_{S2}, S2^I_0, I_2) \wedge N_2 = \text{ite}(w^N_{S2}, S2^N_0, N_1) \wedge \\
&c1_2 = \text{ite}(w^{c1}_{S2}, S2^{c1}_0, c1_1) \wedge c2_2 = \text{ite}(w^{c2}_{S2}, S2^{c2}_0, c2_1) \wedge c3_2 = \text{ite}(w^{c3}_{S2}, S2^{c3}_0, c3_1) \\
&\wedge I_4 = I_3 + 1 \wedge \\
&I_4 \geq N_2
\end{aligned}$$

The encoding (ϕ_2) of the target path is similar.

The boolean variables w_s^v mean that statement s writes to variable v. This is required because $v \in \mathsf{W}(s)$ states that s *may* write to variable v, but it is not mandatory to do so. We define ϕ_w to be the conjunction of the following set of constraints (for each pair of template statement s and variable v):

$$w_s^v \rightarrow v \in \mathsf{W}(s)$$

We now generate the constraints ϕ_u that assert when the fresh values generated from the template statements/expressions are equal. These constraints are akin to Ackermann's reduction for uninterpreted function symbols. For example, to state when B_0 and B_1 are equal, we use the following constraint:

$$\big((I \in \mathsf{R}(\mathsf{B}) \rightarrow I_0 = I_2) \wedge (N \in \mathsf{R}(\mathsf{B}) \rightarrow N_0 = N_1) \wedge (c_1 \in \mathsf{R}(\mathsf{B}) \rightarrow c1_0 = c1_1) \wedge \\ (c_2 \in \mathsf{R}(\mathsf{B}) \rightarrow c2_0 = c2_1) \wedge (c_3 \in \mathsf{R}(\mathsf{B}) \rightarrow c3_0 = c3_1)\big) \rightarrow B_0 = B_1$$

Set membership constraints ($x \in y$) are encoded as boolean variables.

Now that we have all the necessary constraints, we construct the following formula ϕ and give it to an SMT solver:

$$\forall V \ (\phi_1 \wedge \phi_2 \wedge \phi_w \wedge \phi_u \rightarrow \\ I_4 = I_8 \wedge N_2 = N_4 \wedge c1_2 = c1_4 \wedge c2_2 = c2_4 \wedge c3_2 = c3_4)$$

where I_4/I_8, N_2/N_4, $c1_2/c1_4$, $c2_2/c2_4$, and $c3_2/c3_4$ are the final values of the $I/N/c_1/c_2/c_3$ variables of the source and target templates, respectively. V is the set of variables that are universally quantified. These include the variables w_s^v, and all the fresh variables created by the path encoding process.

Giving this formula to an SMT solver will yield assignments to the boolean variables corresponding to the read and write sets' membership (the only existentially quantified variables). Each set of these assignments (a model of the formula) is a sufficient precondition that makes the two paths equivalent (or unreachable).

For this formula, we may obtain the model $\mathsf{R}(\mathsf{B}) = \emptyset \wedge \mathsf{W}(\mathsf{S}_1) = \{c_1\} \wedge \mathsf{W}(\mathsf{S}_2) = \{c_2\}$. Although this condition is certainly sufficient to make the first path unreachable (because it implies that B is a constant, and therefore it cannot evaluate to two different values), this condition is not the weakest.

As we stated before, our algorithm is iterative and so it keeps weakening the counterexample's precondition until it gets the weakest precondition. We do so by negating each model, adding it to formula ϕ, and then retrieving another model from the SMT solver. We stop when there are no more models (i.e., the conjunction of ϕ and the negation of all the previously discovered models is unsatisfiable). The weakest precondition for the counterexample is the disjunction of all models.

After processing one counterexample, we strengthen the transformation function's weakest precondition with the counterexample's weakest precondition. We iterate until there are no more counterexamples, i.e., until the transformation function is correct.

The algorithm terminates because the precondition is strengthened when each counterexample is processed and because the language of preconditions we consider is finite.

Finally, the precondition we obtain for our example (loop unswitching) after processing all the counterexamples is the following:

$$P = I \notin \mathsf{R(B)} \wedge \mathsf{W(S_1)} \cap \mathsf{R(B)} = \emptyset \wedge \mathsf{W(S_2)} \cap \mathsf{R(B)} = \emptyset$$

Optimizations. The algorithm we just presented informally will take significant time to terminate, since it will usually enumerate many models. We present two optimizations that improve the speed of convergence significantly, as well as improve the compactness of the generated preconditions.

The first optimization we perform is model weakening, meaning that given a model generated by an SMT solver, we try to make it weaker (more general) by dropping literals from it (which are therefore "don't cares"). For a model μ of ϕ, we know that $\neg\phi \wedge (\bigwedge l \in \mu)$ is unsatisfiable. Moreover, if for some literal l', $\neg\phi \wedge (\bigwedge l \in \mu' = \mu \setminus \{l'\})$ is still unsatisfiable, then we know that both $\mu' \cup \{l'\}$ and $\mu' \cup \{\neg l'\}$ are models of ϕ. Therefore, μ' is a weaker (partial) model of ϕ. We leverage this knowledge to iterate over each of the literals in a model to check which ones can be removed.

The second optimization we perform is to add additional constraints to ϕ that represent common precondition patterns. In particular, we noticed that stating that the intersection of read/write sets must be empty (e.g., $\mathsf{W(S_1)} \cap \mathsf{R(B)} = \emptyset$) is a common pattern. We therefore associate a boolean variable to each of such constraints, and try to bias the weakening of the models (as previously described) towards these variables. This optimization not only produces more compact preconditions, but also reduces the number of models considerably (since it avoids enumerating all the models that correspond to the constraint they succinctly imply).

4 The Algorithm

Our precondition synthesis algorithm, named PSyCO, is counterexample-guided. The algorithm relies on a verification tool as a black box that can prove the correctness of optimizations (such as CORK [25] or PEC [20]) or return a counterexample otherwise.

4.1 PSyCO

The pseudo-code for the PSyCO algorithm is shown in Figure 3. The algorithm takes as input a transformation function τ and returns the corresponding weakest precondition ψ. Starting with the precondition true, the algorithm iteratively calls the CheckTF function that checks whether τ is correct under the given precondition or returns a counterexample otherwise (given as two paths, π_1 and π_2, of the source and target programs, respectively). The CheckTF function is a black box given as input.

```
function PSyCO
input
  $\tau$ – transformation function
vars
  $\psi$ – generated precondition
begin
  $\psi$ := true
  repeat
     match CheckTF($\tau, \psi$) with
      | correct ->
         return $\psi$
      | counterexample ($\pi_1, \pi_2$) ->
         $\psi := \psi \wedge$ SynthWP($\pi_1, \pi_2$)
end.
```

Fig. 3. PSyCO algorithm

At each step, PSyCO strengthens the precondition with a condition that is sufficient (and necessary) to discharge the counterexample. Therefore, a given counterexample is never seen more than once. Since the number of possible combinations of preconditions in the considered language is finite and we keep strengthening the precondition at each step, we have that PSyCO terminates (assuming that CheckTF always terminates).

4.2 SynthWP

Figure 4 shows the function SynthWP that takes two paths, π_1 and π_2, as input, and returns the weakest precondition that makes the two paths equivalent.

The idea is to construct an universally quantified formula such that a model for it guarantees that the two paths are equivalent (or either one becomes unreachable) for all possible program inputs. The union of all models is the weakest precondition. The models can be generated with an off-the-shelf SMT solver.

SynthWP starts by generating a formula that corresponds to each of the counterexample paths (lines 3 and 4). This is done using standard techniques, that we do not describe here. VCGen takes as input a path π, a map σ_0 with the initial values of the program variables, a set of variables V containing the variables of the source path, and a map w containing a fresh boolean variable for each pair of statements and variables. If a certain statement writes to a given program variable, its corresponding boolean variable in w will be true.

VCGen replaces each template statement s with the following constraint:

$$\bigwedge_{v \in V} (v' = \text{ite}(w(s, v), fv, v))$$

where v' is the new value of v and fv is a fresh variable (one per variable v). Template expressions are replaced with a fresh variable.

VCGen returns a formula corresponding to the input path, a map σ with the final value of each of the program variables and a set u. The set u contains triples

```
    function SynthWP
    input
        $\pi_1, \pi_2$ – counterexample paths
    vars
        $\psi$ – generated precondition
    begin
1       $\sigma_0 := \{v \mapsto \text{fresh integer var} \mid v \in \mathsf{Vars}(\pi_1;\pi_2)\}$
2       $w := \{(s,v) \mapsto \text{fresh boolean var} \mid s \in \mathsf{Stmts}(\pi_1;\pi_2) \wedge v \in \mathsf{Vars}(\pi_1;\pi_2)\}$
3       $\phi_1, \sigma_1, u_1 := \text{VCGen}(\pi_1, \sigma_0, \mathsf{Vars}(\pi_1), w)$
4       $\phi_2, \sigma_2, u_2 := \text{VCGen}(\pi_2, \sigma_0, \mathsf{Vars}(\pi_1), w)$
5       $\phi_u := \bigwedge_{(\sigma,v,t),(\sigma',v',t') \in (u_1 \cup u_2) \wedge t=t'}$
                $\left(\left(\bigwedge_{v'' \in \mathsf{Vars}(\pi_1)} (\mathcal{B}(v'' \in \mathsf{R}(t)) \to \sigma(v'') = \sigma'(v''))\right) \to v = v'\right)$
6       $\phi_w := \bigwedge_{((s,v) \mapsto l) \in w} (l \to \mathcal{B}(v \in \mathsf{W}(s)))$
7       $\phi_c := \bigwedge_{s \in \mathsf{Stmts}(\pi_1;\pi_2)} (\mathcal{B}(\mathsf{CtxVar}(s) \in \mathsf{W}(s)))$
8       $\phi_d, d := \text{MkDisj}(\pi_1;\pi_2)$
9       $V := \{\sigma_0(v) \mid v \in \mathsf{Vars}(\pi_1;\pi_2)\} \cup \{l \mid ((s,v) \mapsto l) \in w\} \cup$
                $\{v \mid (\sigma,v,t) \in (u_1 \cup u_2)\} \cup d$
10      $\phi := \forall V \left(\phi_u \wedge \phi_w \wedge \phi_c \wedge \phi_d \wedge \phi_1 \wedge \phi_2 \to \bigwedge_{v \in \mathsf{Vars}(\pi_1)} (\sigma_1(v) = \sigma_2(v))\right)$
11      $\mu_f := \{\neg\mathcal{B}(v \in \mathsf{R}(t)) \mid v \in \mathsf{Vars}(\pi_1;\pi_2) \wedge t \in \mathsf{Tmpl}(\pi_1;\pi_2)\} \cup$
                $\{\neg\mathcal{B}(v \in \mathsf{W}(s)) \mid v \in \mathsf{Vars}(\pi_1;\pi_2) \wedge s \in \mathsf{Stmts}(\pi_1;\pi_2)\} \cup d$
12      $\psi := \mathsf{false}$
13      while $\phi \wedge \neg\psi$ is satisfiable do
14          $\psi := \psi \vee \text{GeneralizeWP}(\phi, \text{GetModel}(\phi) \cap \mu_f, d)$
15      return $\psi$
    end.
```

Fig. 4. SynthWP algorithm

(σ, v, t), one per each fresh variable v created for template statement/expression t, with σ being a map with the value of the variables at the point where the template statement/expression was evaluated.

In line 5, we generate a formula akin to Ackermann's reduction for uninterpreted function symbols. For each pair of triples (σ, v, t) and (σ', v', t') in w coming from the same template (i.e., $t = t'$), we assert that the fresh variables v and v' must be equal if each of the variables in the read set of t has the same value in σ and σ'. We use the notation $\mathcal{B}(x \in y)$ to introduce a boolean variable that represents that $x \in y$.

In line 6, we generate a constraint that asserts that a template statement can only write to a variable v if v is in its write set. The constraint generated in line 7 asserts that each template statement s_i has at least one distinct context variable c_i in its write set. This is an important optimization, since it avoids the generation of multiple equivalent models that are equal modulo a renaming of the context variables.

In line 8, we introduce a set of boolean variables to represent constraints of the form $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) = \emptyset$ and $\mathsf{R}(t) \cap \mathsf{W}(s_1) = \emptyset$ for every pair of template statements s_1 and s_2 and template statements/expressions t. This is an optimization that enables us to more succinctly express preconditions of this form without having to enumerate all the possible combinations of read and write sets that satisfy the corresponding constraint.

For a path π, MkDisj generates constraints of the form:

$$\bigwedge_{s,s' \in \mathsf{Stmts}(\pi)} \left(\left(\neg \bigvee_{v \in \mathsf{Vars}(\pi)} (\mathcal{B}(v \in \mathsf{W}(s)) \wedge \mathcal{B}(v \in \mathsf{W}(s'))) \right) \leftrightarrow fv \right)$$

with fv being a fresh variable (one per each pair (s, s')). MkDisj generates similar constraints for every pair of read and write sets. In addition to the generated constraint, MkDisj returns the set of fresh variables used.

In line 9, we collect the set of variables that will be universally quantified, namely the set of initial values of the variables and the set of fresh variables used in previous steps. The remaining variables (the booleans representing set membership and the variables in d) are implicitly existentially quantified. Finally, in line 10, we assemble the final constraint. It states that either one of the paths is unreachable or the final value of the variables of the two paths must be equal.

In line 11, we compute a model filter, since we are only interested in negative membership constraints and empty intersection constraints (d). We do not need to consider positive membership constraints, since e.g., $v \in \mathsf{R}(t)$ means that t *may* read v (but not necessarily). Therefore, a model μ including a positive membership constraint l, e.g., $\mu = \mu' \cup \{l\}$, implies that $\mu' \cup \{\neg l\}$ is also a model.

Lines 13–15 implement the main synthesis loop. We iterate over the models of the formula ϕ (filtered by μ_f) and generalize each one in the hope that we will produce more succinct preconditions and converge faster. We pass the set of variables d to GeneralizeWP, so that it can bias the result and express it over more literals of d whenever possible. Function GetModel is given by the SMT solver and returns a model for the formula given as input.

Encoding Size. In the worst-case, the size of formula ϕ is dominated by ϕ_u, since that is the only constraint that grows quadratically with the size of the input. Given a counterexample (π_1, π_2), the worst-case size of ϕ_u (and therefore of ϕ) is $O\left((|\pi_1 \ ; \ \pi_2| \cdot |\mathsf{Vars}(\pi_1)|)^2\right)$.

4.3 GeneralizeWP

Figure 5 shows the function GeneralizeWP. As input, it takes a formula ϕ, a model μ of ϕ given as a set of literals, and a set of preferred literals ψ. The purpose of this function is to compute a new model for ϕ, hopefully smaller than μ (and obviously not bigger), while maximizing the set of literals of ψ that will be part of the result.

From the definition of model of a formula, we know that $\neg\phi \wedge (\bigwedge l \in \mu)$ is unsatisfiable. If we a drop a literal, say l', from μ and if $\neg\phi \wedge (\bigwedge l \in \mu \setminus \{l'\})$ is still unsatisfiable, then $\mu' = \mu \setminus \{l'\}$ is a model of ϕ as well. However, μ' contains fewer literals than μ, and is therefore more generic (since we now know that both $\mu' \cup \{l\}$ and $\mu' \cup \{\neg l\}$ are models of ϕ).

```
    function GeneralizeWP
    input
      $\phi$ – a formula
      $\mu$ – a model of formula $\phi$ (set of literals)
      $\psi$ – set of preferred literals to bias the solution
    begin
1     if $\neg\phi \wedge (\bigwedge l \in \mu \cap \psi)$ is unsatisfiable
2         return MinimizeCore($\neg\phi$, GetUnsatCore($\neg\phi, \mu \cap \psi$))
3     else
4         return MinimizeCore($\neg\phi$, GetUnsatCore($\neg\phi, \mu$) $\cup (\mu \cap \psi)$))
    end.
```

Fig. 5. GeneralizeWP algorithm

```
    function MinimizeCore
    input
      $\phi$ – a formula
      $\zeta$ – an unsat core of formula $\phi$ (set of literals)
    vars
      $\Psi$ – minimized core
    begin
1     $\Psi := \emptyset$
2     while $\zeta \neq \emptyset$ do
3         $\kappa :=$ take one from $\zeta$
4         if $\phi \wedge (\bigwedge l \in \Psi \cup \zeta)$ is satisfiable then
5             $\Psi := \Psi \cup \{\kappa\}$
6     return $\Psi$
    end.
```

Fig. 6. MinimizeCore algorithm

Function GeneralizeWP works as follows. First, it checks whether restricting the model to the set of preferred literals is sufficient to make ϕ unsatisfiable. If so, it calls MinimizeCore to further reduce the size of the solution. We use the function GetUnsatCore, which is usually provided by SMT solvers, as an optimization. GetUnsatCore(x, y) returns a set $y' \subseteq y$ such that $x \wedge (\bigwedge l \in y')$ is unsatisfiable.

If the set of preferred literals is not enough, then we call MinimizeCore with the whole model. Since we are not able to bias the result of GetUnsatCore, we need to ensure that the set of preferred literals is passed to MinimizeCore.

4.4 MinimizeCore

Figure 6 shows the function MinimizeCore. Given a formula ϕ and a set of literals ζ such that $\phi \wedge (\bigwedge l \in \zeta)$ is unsatisfiable, the objective of this function is to find a possibly smaller set $\Psi \subseteq \zeta$ such that $\phi \wedge (\bigwedge l \in \Psi)$ is still unsatisfiable.

MinimizeCore works by checking if each literal $l \in \zeta$ is necessary for the formula to be unsatisfiable. If so, l is added to the result set Ψ. We employ a linear search, as opposed to potentially better search strategies such as QuickXplain [19] or Progression [26], since linear search proved to perform well in our benchmarks.

In our implementation, ζ is a list and we perform a linear search from the beginning to the end of the list. This strategy enables us to bias the search to give priority for removal of certain literals. In particular, in the function GeneralizeWP, we put all the preferred literals ψ at the end of the list, which biases the solution towards having a higher number of literals of ψ.

4.5 Discussion

The proposed algorithm, although agnostic to the verification algorithm used, assumes that only counterexamples for partial functional correctness proofs are generated. This means that the algorithm as presented will produce weakest *liberal* preconditions. To produce weakest preconditions, the algorithm has to be extended so that it can handle counterexamples for relative termination mismatches (based upon, e.g., [5,8,17]).

The proposed specification language does not include instructions to access heap locations or arrays. This means that the current algorithm does not handle optimizations that perform explicit transformations to memory access instructions. It does, however, support instantiation of templates with memory accessing instructions (such as instantiating a template expression with a load from a memory location), provided that the instantiation meets the precondition (which can be verified using, e.g., a data dependency analysis).

In this paper, we only consider preconditions in the language of read and write sets. However, arithmetic preconditions may be needed for some optimizations. For example, a specialization of the loop unrolling optimization requires the number of iterations of the source loop to be even[1]. Synthesizing such preconditions could be done by, for example, adapting the counterexample-driven algorithm of Seghir and Kroening [33].

5 Evaluation

We implemented a prototype named PSyCO[2], which stands for Precondition Synthesizer for Compiler Optimizations. PSyCO is implemented in Python (in about 1,400 lines of code), and uses Z3 4.3.2 [10] for constraint solving.

In principle, PSyCO can be used with any compiler optimization verification tool that can produce counterexamples. However, we chose to implement a simple bounded model checker (BMC) within PSyCO for convenience. This BMC only checks optimizations for partial correctness, and therefore the results we present in this section are weakest *liberal* preconditions. We did not use our own verification tool, CORK [25], since it is several orders of magnitude slower at

[1] We used a more general version of loop unrolling (that accounts for an even and odd number of loop iterations) in our experiments.

[2] Prototype and benchmarks available from `http://web.ist.utl.pt/nuno.lopes/psyco/`.

producing counterexamples than our simple BMC. Furthermore, CORK does not support disjunctive preconditions natively, and therefore it has to first convert such preconditions to DNF and test each of the conjuncts separately.

We show in Table 1 a few examples of compiler optimizations and the corresponding preconditions generated by PSyCO. This list is not supposed to be exhaustive, since there are many optimizations and each one of them may be specified in slightly different ways. We show these examples so that the reader can truly appreciate the simplicity of the generated preconditions and realize how surprising the weakest preconditions can be (from what you would expect at first thought).

There are very few published formally stated preconditions for compiler optimizations. However, the PEC paper [20] does include a precondition for software pipelining (in a slightly different language than the one we used), that was written down by hand and then verified correct by PEC. The precondition synthesized by PSyCO (as shown in Table 1) is, however, weaker than that in PEC's paper. Their precondition requires that $V_1 \notin \mathsf{W}(\mathsf{S}_1)$ and $V_2 \notin \mathsf{W}(\mathsf{S}_1)$, while the precondition generated by PSyCO does not. Therefore, the precondition generated by PSyCO is weaker than that published by experts in formal methods and compiler optimizations, showing that automatic precondition synthesis does indeed help to make optimizations more widely applicable.

We ran PSyCO over a set of optimizations (mostly loop manipulating). The experiments were run on a machine running Linux 3.10.10 with an Intel Core 2 Duo 3.00 GHz CPU, and 4 GB of RAM. The results are shown in Table 2.

Since we are not aware of any other algorithm for precondition synthesis for compiler optimization, we cannot compare PSyCO against other tools. In Table 2, we show the number of counterexamples required for each optimization to reach convergence. We notice that in general only a few counterexamples are required (with certain optimizations with a trivial precondition requiring none).

Then, we present the total number of models obtained for preconditions. The ratio of the number of models per number of counterexamples is small because of the employed optimizations described before (model weakening and inference of common precondition patterns).

Finally, we show the time taken by the precondition synthesis algorithm, as well as the overall time taken by the tool. The overall time includes not only the synthesis algorithm, but also the BMC time as well as minor initializations performed by the tool. We argue that the time taken by the precondition synthesis algorithm is low. Overall, PSyCO is usually fast, with a few exceptions due to high inefficiencies in the Z3Py module exposed by our BMC. However, the running time for this kind of tool is not critical, since it is supposed to be run off-line, and only once per optimization specification.

6 Related Work

This work is related with both precondition synthesis and compiler (optimizations) correctness. We briefly describe both topics here.

Table 1. Examples of weakest preconditions synthesized by PSyCO

Optimization	Weakest Liberal Precondition
Partial redundancy elimination (PRE): **if** B **then** S_1; $V_1 := \mathsf{E}$; S_2 **else** S_3; $V_2 := \mathsf{E}$ $\Rightarrow$ **if** B **then** S_1; $V_1 := \mathsf{E}$; S_2; $V_2 := V_1$ **else** S_3; $V_2 := \mathsf{E}$	$V_1 \notin \mathsf{R}(\mathsf{E}) \wedge V_1 \notin \mathsf{W}(\mathsf{S}_2) \wedge \mathsf{R}(\mathsf{E}) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset$
Code hoisting: **if** B **then** S_1; S_2 **else** S_1; S_3 $\Rightarrow$ S_1; **if** B **then** S_2 **else** S_3	$\mathsf{R}(\mathsf{B}) \cap \mathsf{W}(\mathsf{S}_1) = \emptyset$
Loop unrolling: **while** $V_1 < V_2$ **do** S; $V_1 := V_1 + 1$ $\Rightarrow$ **while** $(V_1 + 1) < V_2$ **do** S; $V_1 := V_1 + 1$; S; $V_1 := V_1 + 1$ — **if** $V_1 < V_2$ **then** S; $V_1 := V_1 + 1$	$V_2 \notin \mathsf{W}(\mathsf{S}) \wedge (V_1 \notin \mathsf{W}(\mathsf{S}) \vee \mathsf{R}(\mathsf{S}) \cap \mathsf{W}(\mathsf{S}) = \emptyset)$
Strength reduction: **while** $V_1 < V_2$ **do** $V_3 := V_1 * \mathsf{E}$; S; $V_1 := V_1 + 1$ $\Rightarrow$ $V_4 := V_1 * \mathsf{E}$; **while** $V_1 < V_2$ **do** $V_3 := V_4$; $V_4 := V_4 + \mathsf{E}$; S; $V_1 := V_1 + 1$	$V_1 \notin \mathsf{R}(\mathsf{E}) \wedge V_3 \notin \mathsf{R}(\mathsf{E}) \wedge \mathsf{R}(\mathsf{E}) \cap \mathsf{W}(\mathsf{S}) = \emptyset \wedge \big(V_1 \notin \mathsf{W}(\mathsf{S}) \vee (V_3 \notin \mathsf{R}(\mathsf{S}) \wedge \mathsf{R}(\mathsf{S}) \cap \mathsf{W}(\mathsf{S}) = \emptyset)\big)$
Software pipelining: **while** $V_1 < V_2$ **do** S_1; S_2; $V_1 := V_1 + 1$ $\Rightarrow$ **if** $V_1 < V_2$ **then** S_1; **while** $V_1 < (V_2 - 1)$ **do** S_2; $V_1 := V_1 + 1$; S_1 — S_2; $V_1 := V_1 + 1$	$V_2 \notin \mathsf{W}(\mathsf{S}_2) \wedge \big((\mathsf{R}(\mathsf{S}_1) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset \wedge \mathsf{R}(\mathsf{S}_1) \cap \mathsf{W}(\mathsf{S}_1) = \emptyset \wedge \mathsf{R}(\mathsf{S}_2) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset) \vee V_1 \notin \mathsf{W}(\mathsf{S}_2)\big)$

Table 2. List of compiler optimizations [1,28], the number of counterexamples processed, the number of models obtained for preconditions, the time taken by the precondition generation algorithm, and the overall time taken by the tool (including the BMC)

Optimization	# Counterexamples	# Models	WP Time	Total Time
Code hoisting	1	1	0.07s	0.23s
Constant propagation	1	1	0.04s	0.16s
Copy propagation	0	0	0s	0.11s
If-conversion	0	0	0s	0.11s
Partial redundancy elimin.	1	1	0.10s	0.30s
Loop fission	6	36	1.28s	2.18s
Loop flattening	1	1	0.07s	3.31s
Loop fusion	6	36	1.26s	2.19s
Loop interchange	11	25	1.42s	23.8s
Loop invariant code motion	3	3	0.22s	0.55s
Loop peeling	0	0	0s	0.27s
Loop reversal	4	7	0.25s	0.54s
Loop skewing	1	1	0.06s	163s
Loop strength reduction	1	2	1.14s	1.41s
Loop tiling	1	1	0.07s	4.60s
Loop unrolling	2	4	0.13s	0.50s
Loop unswitching	2	2	0.15s	0.77s
Software pipelining	1	2	0.13s	0.58s

6.1 Precondition Synthesis

The concepts of weakest preconditions (WPs) and weakest liberal preconditions (WLPs) have long been introduced by Dijkstra [11]. Since then, several algorithms have been published to accomplish their automatic generation.

There are several competing approaches for WLP synthesis. These include, for example, precondition templates and constraint solving (e.g., [15]), quantifier elimination (e.g., [27]), abstract interpretation (e.g., [9]), and CEGAR, predicate abstraction, and interpolation for predicate generation (e.g., [33]). Some algorithms combine multiple techniques to achieve better performance.

Our unsat core minimization algorithm, that biases the result towards certain literals, is similar to the one presented by Seghir and Kroening [33].

Leino [21] describes a compact encoding for verification conditions generated from the weakest precondition calculus.

Cook et al. [8] propose a counterexample-driven algorithm for precondition synthesis (not necessarily weakest) to guarantee program termination, and Bozga et al. [5] propose an algorithm based on abstract interpretation.

Calcagno et al. [7] present an algorithm for WLP synthesis based on separation logic.

Gulwani et al. [14] present an algorithm to synthesize loop-free programs that implement a given specification. While the goal of the algorithm is not to synthesize preconditions, there is a similarity in the encoding of program equivalence and in the usage of an SMT solver to find assignments to variables that represent the synthesized artifact.

6.2 Compiler Correctness

Several approaches have been proposed to improve the correctness of compilers, including manual and computer-assisted proofs, automatic verification, and automatic generation of correct optimizations by construction.

CompCert [24] is a compiler that aims to provide end-to-end correctness guarantees (from a program's source code down to the resulting binary). CompCert was written from scratch with verification in mind, and its correctness proofs are done in Coq. Vellvm [41] is a Coq-based framework that enables the development and verification of compiler optimizations for LLVM.

CORK [25] is a compiler optimization verifier based on recurrence computation. PEC [20, 37] (a successor of Cobalt [22] and Rhodium [23]) is also a verifier, but uses bisimulation relation synthesis as the underlying technique. Both CORK and PEC require a precondition to be given as input.

Translation validation (e.g., [13, 30, 31, 34, 38, 40, 42]) is a technique for establishing the correctness of compiler optimizations *after* the optimization was run by checking the original and optimized programs for equivalence. Namjoshi and Zuck [29] propose augmenting transformation functions so that they generate auxilarly invariants to help the translation validation process, which otherwise could fail to derive those invariants automatically.

Guo and Palsberg [16] present a bisimulation-based technique to reason about the correctness of trace optimizations.

Godlin and Strichman [12] propose a set of proof rules to prove equivalence of programs and to prove mutual termination using uninterpreted function symbols to abstract recursive function calls. The technique is later extended with the introduction of mutual summaries [17].

Relational Hoare logic [4] is an extension to Hoare logic to prove equivalence of programs. Barthe et al. [3] extend this work to support non-structurally equivalent programs.

Superoptimization (e.g., [2, 6, 18, 35]) is a technique to do code optimization given a set of theorems that establish equalities between code sequences and then searching for a better equivalent program.

Tate et al. [36] propose an algorithm to extrapolate compiler optimizations directly from concrete examples.

Scherpelz et al. [32] propose an algorithm to automatically synthesize flow functions from compiler optimizations' preconditions.

7 Conclusion

In this paper we presented, to the best of our knowledge, the first algorithm for the automatic synthesis of weakest preconditions for compiler optimizations. The algorithm generates preconditions iteratively, in a counterexample-driven approach. Preconditions for counterexamples are generated by an SMT solver.

We built a prototype, named PSyCO, that implements the proposed algorithm and we presented several preconditions generated by it.

Acknowledgments. The authors thank the anonymous reviewers for their comments and suggestions on earlier drafts of this paper.

This work was partially supported by the FCT grants SFRH/BD/63609/2009 and INESC-ID multiannual funding PEst-OE/EEI/LA0021/2013.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley (2006)
2. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: ASPLOS (2006)
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL (2004)
5. Bozga, M., Iosif, R., Konečný, F.: Deciding conditional termination. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 252–266. Springer, Heidelberg (2012)
6. Brain, M., Crick, T., De Vos, M., Fitch, J.: TOAST: Applying answer set programming to superoptimisation. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 270–284. Springer, Heidelberg (2006)
7. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
8. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
9. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 128–148. Springer, Heidelberg (2013)
10. 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)
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
12. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Inf. 45(6), 403–439 (2008)
13. Goldberg, B., Zuck, L., Barrett, C.: Into the loops: Practical issues in translation validation for optimizing compilers. Electron. Notes Theor. Comp. Sci. 132 (2005)
14. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011)
15. 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)
16. Guo, S.-Y., Palsberg, J.: The essence of compiling with traces. In: POPL (2011)
17. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards modularly comparing programs using automated theorem provers. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 282–299. Springer, Heidelberg (2013)
18. Joshi, R., Nelson, G., Zhou, Y.: Denali: A practical algorithm for generating optimal code. ACM Trans. Program. Lang. Syst. 28(6), 967–989 (2006)
19. Junker, U.: QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In: AAAI (2004)

20. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: PLDI (2009)
21. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. 93(6), 281–288 (2005)
22. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: PLDI (2003)
23. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: POPL (2005)
24. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
25. Lopes, N.P., Monteiro, J.: Automatic equivalence checking of UF+IA programs. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 282–300. Springer, Heidelberg (2013)
26. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 592–607. Springer, Heidelberg (2013)
27. 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)
28. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
29. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013)
30. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000)
31. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
32. Scherpelz, E.R., Lerner, S., Chambers, C.: Automatic inference of optimizer flow functions from semantic meanings. In: PLDI (2007)
33. Seghir, M.N., Kroening, D.: Counterexample-guided precondition inference. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 451–471. Springer, Heidelberg (2013)
34. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 737–742. Springer, Heidelberg (2011)
35. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL (2009)
36. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Generating compiler optimizations from proofs. In: POPL (2010)
37. Tatlock, Z., Lerner, S.: Bringing extensibility to verified compilers. In: PLDI (2010)
38. Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: PLDI (2011)
39. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI (2011)
40. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008)
41. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: PLDI (2013)
42. Zuck, L., Pnueli, A., Goldberg, B., Barrett, C., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Form. Methods Syst. Des. 27 (2005)

Message-Passing Algorithms for the Verification of Distributed Protocols

Loïg Jezequel and Javier Esparza

Institut für Informatik, Technische Universität München, Germany

Abstract. Message-passing algorithms (MPAs) are an algorithmic paradigm for the following generic problem: given a system consisting of several interacting components, compute a new version of each component representing its behaviour inside the system. MPAs avoid computing the full state space by propagating messages along the edges of the system interaction graph. We present an MPA for verifying local properties of distributed protocols with a tree communication structure. We report on an implementation, and validate it by means of two case studies, including an analysis of the PGM protocol.

Introduction

Message-passing algorithms (MPAs) are an algorithmic paradigm for problems (called reduction problems) that can be generically described as follows. The input to the problem is a system consisting of several components communicating in some way. When considered in isolation, each component has a set of behaviours. However, not all these behaviours are necessarily realizable within the system, since some actions may need the cooperation of other components. The problem consists of computing a new version of each component whose behaviours are those behaviours of the original component that are realizable within the system.

MPAs work by propagating messages containing information about the behaviour of parts of the system along the edges of its interaction graph. Before sending a message, a component can process it to remove redundant or useless information. This way MPAs avoid computing the full state space. MPAs can be applied on any system, however they ensure to solve the reduction problem only for systems whose interaction graph is a tree. A generic description of MPAs can be found in [1]. In particular, MPAs for distributed planning of [2, 3] have been developed in an automata theoretic setting. More precisely, to each component of the system is attached a (weighted) automaton, and interaction between components is modelled by means of automata-theoretic operations. The nice experimental results obtained with this approach on planning problems suggest to study the use of MPAs for solving more generic formal verification problems, and in this paper we explore this idea.

We present an MPA for verifying local properties of distributed protocols with a tree communication structure. Loosely speaking, "local" means that the property is defined from the point of view of one of the components of the protocol.

For instance, a local property of a protocol for broadcast communication is that a receiver gets all messages sent by the source. On the other hand, the mutual exclusion property for a mutal exclusion protocol with N processes is an example of a global property (which may sometimes be equivalent to a local property, but not always).

We model components as labeled transition systems (LTSs) $\mathcal{L}_1, \ldots, \mathcal{L}_n$, communicating à la CSP by rendez-vous. We present a generic MPA parametrized by an equivalence relation $\equiv$, which is assumed to be a congruence with respect to parallel composition and hiding. The MPA computes LTSs $\mathcal{K}_1, \ldots, \mathcal{K}_n$ such that $\mathcal{K}_i \equiv (\mathcal{L}_1 || \cdots || \mathcal{L}_n) \setminus \overline{\Sigma_i}$, where $\overline{\Sigma_i}$ is the complement of the alphabet of $\mathcal{L}_i$ (that is, we hide all actions but those involving the i-th component). We call $\mathcal{K}_i$ the *update* of $\mathcal{L}_i$. For a system with n-components the MPA requires $2n-2$ messages, which we show is optimal. We then present two different instances of the algorithm, suitable for checking safety and liveness properties, respectively. The first instance instantiates $\equiv$ with the standard trace semantics. The second one chooses for $\equiv$ the semantics consisting of the infinite traces of the LTS plus its set of divergences, i.e., the traces after which an infinite sequence of silent transitions can occur. For both semantics we report on an implementation of the MPA, and we describe the technique used to reduce the size of the messages.

We evaluate our two instances on two case studies: a mutual exclusion protocol for tree networks proposed in [4], and a version of the Pragmatic General Multicast Protocol (PGM) [5]. For the PGM we show how the result of the MPA allows us to identify potential problems of the protocol when the different parameters of the protocol are assigned unsuitable values.

Related Work. The connection of our work to other work on MPAs has been described above.

Several compositional approaches to verification exist, aiming at avoiding the state-explosion problem in the verification of distributed systems by considering them component by component. In [6] the authors use an assume-guarantee way of reasoning in which they show that a component guarantees some property as soon as it is in a system satisfying some assumption, the hard part being to choose good assumptions, which is achieved by progressively learning them. In [7] the authors introduce thread-modular model checking where the states of a multithreaded software are enumerated thread by thread (a state taking into account the value of the program counter of a single thread instead of the value of the program counters of all threads), potentially leading to an exponential reduction of the size of the state space considered, at the price of doing incomplete verification.

Our work is based on the possibility to replace a component of a system by an equivalent one (with respect to a suitable notion of equivalence or preorder), but smaller in some way. This is also the starting point of works such as [8, 9] for example. This approach has been implemented for trace, failures, and bisimulation equivalence or related preorders in tools like FDR [10, 11], the Concurrency Workbench [12], CADP [13] and others, and most model-checkers using abstraction techniques use it in some way. However, these tools address the problem of computing an update of the whole system, instead of an update for each component, and leave the choice of which components to minimize or reduce with respect to the given congruence, and in which order, to the user. (In particular, this hinders a direct comparison with these tools, since we would have to compare a fully automatic and a partially manual procedure. On the other hand, our algorithm could also be implemented on top of these tools.) By focusing on this problem we are able to provide a simple algorithm with a minimal number of exchanged messages.

Protocols with tree communication structure have also been analyzed by means of regular model checking (see [14] for a recent survey). The goal of regular model checking is more ambitious than ours, since it aims at proving the protocol correct for an arbitrary number of processes. On the other hand, this reduces the range of protocols that can be verified. In particular, we do not know of any analysis of our two case studies using regular model checking.

The PGM protocol has been analysed in a number of papers [15–17]. This work is orthogonal to ours. In [15, 16] the focus is on timing aspects and relations between parameters, which are analysed for small instances of the protocol (below five processes), while we concentrate on analyzing simpler properties of larger instances with more than one hundred processes. Finally, the work in [17] is a manual proof providing cut-off bounds for parametric analysis.

1 Definitions and Notations

LTS. A *labelled transition system with silent transitions* (LTS) is a tuple $\mathcal{L} = (\Sigma, S, T, s^0)$ where Σ is a finite set of *labels*, S is a finite set of *states*, $T = \{T^\sigma : \sigma \in \Sigma\} \cup \{T^\tau\}$ is a set of *transition relations*: $\forall \sigma \in \Sigma \cup \{\tau\}, T^\sigma \subseteq S \times S$ (the elements of T^σ are called *σ-transitions*), and $s^0 \in S$ is an *initial state.* A finite (resp. infinite) sequence of labels and τ, $tr = \sigma_1\sigma_2 \ldots$ is a *trace* of $\mathcal{L}$ if there exits a finite (resp. infinite) alternating sequence of states and transitions (a *path*) $\pi = s_0t_1s_1t_2 \ldots$ *realizing* it, that is, such that $s^0 = s_0$, and $t_i = (s_{i-1}, s_i) \in T^{\sigma_i}$ for all $i > 0$. A finite (resp. infinite) sequence of labels tr is an *observable trace* of $\mathcal{L}$ if there exists a trace tr' of $\mathcal{L}$ such that by removing all τ from tr' one gets tr: $tr'|_\Sigma = tr$. The set of finite observable traces of $\mathcal{L}$ is denoted by $\mathcal{T}^*_\mathcal{L}$ while its set of infinite observable traces is denoted by $\mathcal{T}^\omega_\mathcal{L}$, and the set of all its observable traces is $\mathcal{T}_\mathcal{L} = \mathcal{T}^*_\mathcal{L} \cup \mathcal{T}^\omega_\mathcal{L}$. We write $s_0 \overset{\tau}{\rightsquigarrow}_\mathcal{L} s_n$ if there exists a path $\pi = s_0t_1s_1t_2 \ldots s_n$ such that $\forall 1 \leq i \leq n, t_i \in T^\tau$. Figure 1 gives some examples of LTSs, states are represented by circles and labelled transitions by labelled arrows between states. Initial states are distinguished with small arrows.

Definition 1. *The* hiding *of a set Σ' in an LTS $\mathcal{L} = (\Sigma, S, T, s^0)$ is the LTS $\mathcal{L} \setminus \Sigma' = (\Sigma \setminus \Sigma', S, T', s^0)$ with T' such that $T'^\sigma = T^\sigma$ for any $\sigma \in \Sigma \setminus \Sigma'$ and $T'^\tau = \cup_{\sigma \in \Sigma' \cup \{\tau\}} T^\sigma$.*

For $\mathcal{L} = (\Sigma, S, T, s^0)$ and some set of labels Σ' we sometimes write $\mathcal{L} \setminus \overline{\Sigma'}$ for $\mathcal{L} \setminus (\Sigma \setminus \Sigma')$.

Definition 2. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs where $\mathcal{L}_i = (\Sigma_i, S_i, T_i, s_i^0)$. Their* parallel composition *$\mathcal{L}_1 || \mathcal{L}_2$ is the LTS $\mathcal{L} = (\Sigma, S, T, s^0)$ such that: $\Sigma = \Sigma_1 \cup \Sigma_2$, $S = S_1 \times S_2$, if $\sigma \in \Sigma_1 \cap \Sigma_2$ then $T^\sigma = \{((s_1, s_2), (s_1', s_2')) \;:\; (s_1, s_1') \in T_1^\sigma \wedge (s_2, s_2') \in T_2^\sigma\}$, if $\sigma \in \Sigma_1 \setminus \Sigma_2$ then $T^\sigma = \{((s_1, s_2), (s_1', s_2)) \;:\; (s_1, s_1') \in T_1^\sigma \wedge s_2 \in S_2\}$, if $\sigma \in \Sigma_2 \setminus \Sigma_1$ then $T^\sigma = \{((s_1, s_2), (s_1, s_2')) \;:\; s_1 \in S_1 \wedge (s_2, s_2') \in T_2^\sigma\}$, and $T^\tau = \{((s_1, s_2), (s_1', s_2)) \;:\; (s_1, s_1') \in T_1^\tau \wedge s_2 \in S_2\} \cup \{((s_1, s_2), (s_1, s_2')) \;:\; s_1 \in S_1 \wedge (s_2, s_2') \in T_2^\tau\}$, and finally $s^0 = (s_1^0, s_2^0)$.*

Notice that this parallel composition is commutative and associative, so one can safely write $\mathcal{L} = \mathcal{L}_1 || \ldots || \mathcal{L}_n$ for the parallel composition of more than two LTSs. The right LTS in Figure 3 is the parallel composition of the middle LTS in the same figure and the right LTS in Figure 1.

Definition 3. *An equivalence relation between LTSs is called a* congruence for LTSs, *denoted by $\equiv$, if for any LTSs $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}$ such that $\mathcal{L}_1 \equiv \mathcal{L}_2$ and any set of labels Σ one has $\mathcal{L}_1 || \mathcal{L} \equiv \mathcal{L}_2 || \mathcal{L}$ and $\mathcal{L}_1 \setminus \Sigma \equiv \mathcal{L}_2 \setminus \Sigma$.*

MLTS. A *marked labelled transition system* (MLTS), or LTS with marked states, is a tuple $\mathcal{ML} = (\mathcal{L}, F)$ where $\mathcal{L} = (\Sigma, S, T, s^0)$ is an LTS and $F \subseteq S$ is a set of *marked states*. According to the set of marked states one can define the set of *marked traces* $\mathcal{MT}_{\mathcal{ML}}$ of $\mathcal{ML}$ as the set of traces for which there exists a realization verifying some condition on the marked states (examples are given below: automata and Büchi automata).

Definition 4. *Given MLTSs $\mathcal{ML}_1, \mathcal{ML}_2$ where $\mathcal{ML}_i = (\mathcal{L}_i, F_i)$, their* parallel composition *$\mathcal{ML}_1 || \mathcal{ML}_2$ is the MLTS $\mathcal{ML} = (\mathcal{L}, F)$ such that $\mathcal{L} = \mathcal{L}_1 || \mathcal{L}_2$ and $F = F_1 \times F_2$.*

FSA. A *finite state automaton* (FSA) is an MLTS $\mathcal{A} = (\mathcal{L}, F)$ such that a trace *tr* of $\mathcal{A}$ is a marked trace if and only if it is finite and it has a realization $\pi = s_0 t_1 s_1 \ldots s_k$ such that $s_k \in F$. The set $\mathcal{MT}_{\mathcal{A}}$ is usually called the language of $\mathcal{A}$.

NBA. A *Büchi automaton* (NBA) is an MLTS $\mathcal{B} = (\mathcal{L}, F)$ such that a trace *tr* of $\mathcal{B}$ is a marked trace if and only if it is infinite and it has a realization $\pi = s_0 t_1 s_1 t_2 \ldots$ such that there exists an infinite number of $i \geq 0$ for which $s_i \in F$. As for FSAs, the set $\mathcal{MT}_{\mathcal{B}}$ is usually called the language of $\mathcal{B}$.

2 Message-Passing Algorithms

Before presenting a formal description of message-passing algorithms, we illustrate them on an example. Consider the three LTSs of Figure 1. They represent an abstract view of a small distributed system involving three processes: a sender (S), a capacity one channel (C), and a receiver (R). S has to accomplish some task. It initially does a choice between doing it alone (right transition) or doing it together with R (left transition) by exchanging some messages through C.

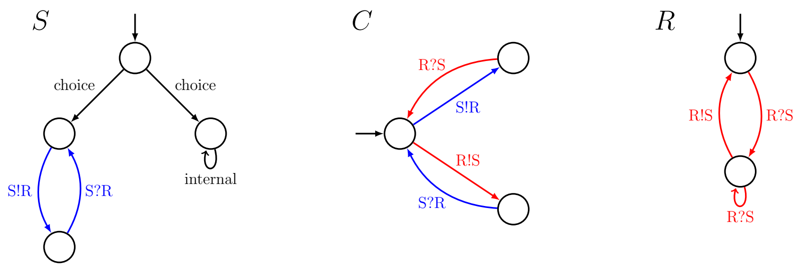


Fig. 1. A distributed system constituted of three interacting LTSs

The behaviour of this system is captured by $\mathcal{L} = S||C||R$, and so – denoting by Σ_S (resp. Σ_C, Σ_R) the set of labels of S (resp. C, R) – the behaviour of S (resp. C, R) inside this system is captured by $\mathcal{L} \setminus \overline{\Sigma}_S$ (resp. $\mathcal{L} \setminus \overline{\Sigma}_C$, $\mathcal{L} \setminus \overline{\Sigma}_R$).

Definition 5. *The* interaction graph *of a system $\mathcal{L}_1|| \cdots ||\mathcal{L}_n$, where Σ_i is the alphabet of $\mathcal{L}_i$, has $\mathcal{L}_1, \ldots, \mathcal{L}_n$ as nodes, and an edge $\{\mathcal{L}_i, \mathcal{L}_j\}$ when $\Sigma_i \cap \Sigma_j \neq \emptyset$.*

The MPAs can solve the reduction problems for systems whose interaction graph is a tree. They proceed by sending messages (which have the same type as the components, i.e. LTSs in our example) along the edges of the tree, i.e. each component sends a message to each of its neighbours. In the system of Figure 1, the interaction graph is a line (and so a tree) with C in the middle and S and R at the extremities (see Figure 2). Indeed S interacts (that is, shares labels) only with C and R also interacts only with C. So each of S and R will send a message to C and C will send a message to each of S and R.

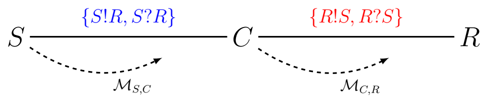


Fig. 2. The interaction graph of the system of Figure 1. Over each edge (plain line) the corresponding set of shared labels is indicated. Dashed lines represent the messages propagated from S to R.

The idea behind these messages is the following. In a tree shaped interaction graph, each edge separates the graph into two subtrees whose roots are the extremities of the edge (in our example, the edge (C, R) separates the graph into a tree containing only R and a tree with C as root and S as leaf). Using this fact each component at the extremity of the removed edge will send a message to the other component. This message describes the possible behaviours of the subtree from which its sender is the root as they can be seen by its receiver (for example C sends a message to R describing the behaviour of $S||C$ as R sees it, that is using only labels shared between C and R). This message is computed from the messages received from the neighbours of its sender in the subtree from which this sender is the root.

In the system of Figure 1 the message from S to C is then $\mathcal{M}_{S,C} \equiv S \setminus \overline{\Sigma_S \cap \Sigma_C}$ (see Figure 3, left for an example preserving observable traces) and it is then used to build the message from C to R: $\mathcal{M}_{C,R} \equiv (\mathcal{M}_{S,C} || C) \setminus \overline{\Sigma_C \cap \Sigma_R}$ (see Figure 3, middle). Similarly the remaining messages can be built: $\mathcal{M}_{R,C} \equiv R \setminus \overline{\Sigma_R \cap \Sigma_C}$ and $\mathcal{M}_{C,S} \equiv (\mathcal{M}_{R,C} || C) \setminus \overline{\Sigma_C \cap \Sigma_S}$. It can then be proved (it is a consequence of Theorem 1 below) using the separation property of the edges of a tree shaped interaction graph described above, that the composition of each component with all the messages it received describes the behaviour of this component in the full system. For example (see Figure 3, right) $R' = R || \mathcal{M}_{C,R}$ has the same set of observable traces than $\mathcal{L} \setminus \overline{\Sigma}_R$. Similarly $S' = S || \mathcal{M}_{C,S}$ and $C' = C || \mathcal{M}_{R,C} || \mathcal{M}_{S,C}$ have the same set of observable traces than $\mathcal{L} \setminus \overline{\Sigma}_S$ and $\mathcal{L} \setminus \overline{\Sigma}_C$ respectively.

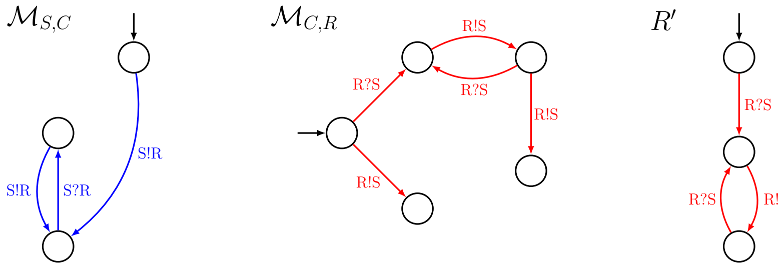


Fig. 3. Messages from S to R, updated component R'

2.1 Formal Description of an MPA for LTSs

Algorithm 1 below presents a formal description of an MPA for a system $\mathcal{L} = \mathcal{L}_1 || \ldots || \mathcal{L}_n$ whose interaction graph $\mathcal{G} = (V, E)$ is a tree. Any LTS $\mathcal{K}_i$ obtained at the end of Algorithm 1 is called the *update* of $\mathcal{L}_i$ (in $\mathcal{G}$).

For the presentation it is convenient to model an undirected edge $\{\mathcal{L}_i, \mathcal{L}_j\}$ as two directed edges $(\mathcal{L}_i, \mathcal{L}_j)$ and $(\mathcal{L}_j, \mathcal{L}_i)$. So the input to Algorithm 1 is a directed graph $\mathcal{G}$ derived from a tree in this way.

We denote by $x :\equiv \mathcal{L}$ that the variable x is assigned some LTS $\mathcal{L}'$ such that $\mathcal{L}' \equiv \mathcal{L}$. Notice that this is a nondeterministic assignment, and so Algorithm 1 is nondeterministic. In the next section we present several instances of the algorithm, and for each one we explain how the nondeterminism is resolved.

In order to formulate and prove the correctness of the algorithm we introduce some notations.

Definition 6. *Given $\mathcal{G} = (V, E)$ with $V = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, we denote the parallel composition $(\mathcal{L}_1 || \cdots || \mathcal{L}_n)$ by $\widehat{\mathcal{G}}$.*
Given a tree $\mathcal{G} = (V, E)$ and $(\mathcal{L}_i, \mathcal{L}_j) \in E$, we denote by $\mathcal{G}_{ij}$ the maximal subtree of $\mathcal{G}$ containing $\mathcal{L}_j$ but not $\mathcal{L}_i$.

Algorithm 1. An MPA for LTSs

```
Input: an interaction graph $\mathcal{G} = (V, E)$ with $V = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$
1: $\mathsf{M} \leftarrow E$
2: while $\mathsf{M} \neq \emptyset$ do
3:     choose $(\mathcal{L}_i, \mathcal{L}_j) \in \mathsf{M}$ such that $(\mathcal{L}_k, \mathcal{L}_i) \notin \mathsf{M}$ for every $k \neq j$
4:     $\mathcal{M}_{i,j} :\equiv (\mathcal{L}_i \mid\mid (\mid\mid_{k \neq j, (\mathcal{L}_k, \mathcal{L}_i) \in E} \mathcal{M}_{k,i})) \setminus \overline{\Sigma}_j$
5:     remove $(\mathcal{L}_i, \mathcal{L}_j)$ from $\mathsf{M}$
6: end while
7: for all $i \in V$ do
8:     $\mathcal{K}_i :\equiv \mathcal{L}_i \mid\mid (\mid\mid_{(\mathcal{L}_j, \mathcal{L}_i) \in E} \mathcal{M}_{j,i})$
9: end for
```

Lemma 1. *Let $\mathcal{G} = (V, E)$ be a tree and $(\mathcal{L}_i, \mathcal{L}_j) \in E$. Let $\mathcal{M}^{\mathcal{G}}_{j,i}$ be the content of variable $\mathcal{M}_{j,i}$ after termination of Algorithm 1 on input $\mathcal{G}$. Then $\mathcal{M}^{\mathcal{G}}_{j,i} = \widehat{\mathcal{G}}_{ij} \setminus \overline{\Sigma}_i$.*

Proof. The proof is by induction on the depth of $\mathcal{G}^{ij}$. If $\mathcal{G}^{ij}$ has depth 1, then $\mathcal{G}_{ij}$ contains the vertex $\mathcal{L}_j$ and no arcs. By line 4 we get $\mathcal{M}^{\mathcal{G}}_{j,i} \equiv \mathcal{L}_i \setminus \overline{\Sigma}_i = \widehat{\mathcal{G}}_{ij} \setminus \overline{\Sigma}_j$ (recall that, since we represent undirected trees as directed graphs, we have $(\mathcal{L}_j, \mathcal{L}_i) \in E$).

Assume now that $\mathcal{G}^{ij}$ has depth larger than 1. Let $\mathcal{L}_{j_1}, \ldots, \mathcal{L}_{j_m}$ be the neighbours of $\mathcal{L}_j$ in $\mathcal{G}_{ij}$. Then the trees $\mathcal{G}jj_1, \ldots \mathcal{G}jj_m$ are proper subtrees of $\mathcal{G}ij$, and in particular have smaller depth. By induction hypothesis we have

$$\mathcal{M}^{\mathcal{G}_{ij}}_{j_1,j} \equiv \widehat{\mathcal{G}}_{jj_1} \setminus \overline{\Sigma}_j \quad \ldots \quad \mathcal{M}^{\mathcal{G}_{ij}}_{j_m,j} \equiv \widehat{\mathcal{G}}_{jj_m} \setminus \overline{\Sigma}_j \tag{1}$$

By line 4 of the algorithm, and since $\equiv$ is a congruence, we get

$$\begin{aligned}
\mathcal{M}^{\mathcal{G}}_{j,i} &\equiv (\ \mathcal{L}_j \mid\mid (\mid\mid_{k \neq i, (\mathcal{L}_k, \mathcal{L}_j) \in E} \mathcal{M}^{\mathcal{G}}_{k,j})\ ) \setminus \overline{\Sigma}_i & (2)\\
&\equiv (\ \mathcal{L}_j \mid\mid \mathcal{M}^{\mathcal{G}}_{j_1,j} \mid\mid \cdots \mid\mid \mathcal{M}^{\mathcal{G}}_{j_m,j}\ ) \setminus \overline{\Sigma}_i & (3)\\
&\equiv (\ \mathcal{L}_j \mid\mid \mathcal{M}^{\mathcal{G}_{ij}}_{j_1,j} \mid\mid \cdots \mid\mid \mathcal{M}^{\mathcal{G}_{ij}}_{j_m,j}\ ) \setminus \overline{\Sigma}_i & (4)\\
&\equiv (\ \mathcal{L}_j \mid\mid (\widehat{\mathcal{G}}_{jj_1} \setminus \overline{\Sigma}_j) \mid\mid \cdots \mid\mid (\widehat{\mathcal{G}}_{jj_m} \setminus \overline{\Sigma}_j)\ ) \setminus \overline{\Sigma}_i & (5)
\end{aligned}$$

where (4) follows from (3) because $\mathcal{L}_k$ is a node of $\mathcal{G}_{ij}$ for every $\mathcal{M}_{k,j}$.

Since $\mathcal{G}$ is a tree, the sets of labels of $\widehat{\mathcal{G}}_{jj_1}, \ldots, \widehat{\mathcal{G}}_{jj_m}$ are pairwise disjoint, and so

$$\mathcal{L}_j \mid\mid (\widehat{\mathcal{G}}_{jj_1} \setminus \overline{\Sigma}_j) \mid\mid \cdots \mid\mid (\widehat{\mathcal{G}}_{jj_m} \setminus \overline{\Sigma}_j) \equiv (\mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} \mid\mid \cdots \mid\mid \widehat{\mathcal{G}}_{jj_m}) \setminus \overline{\Sigma}_j \tag{6}$$

Moreover, for the same reason, there are no edges between any of $\mathcal{L}_{j_1}, \ldots, \mathcal{L}_{j_m}$ and $\mathcal{L}_i$. So $\Sigma_i \cap (\Sigma_{j_1} \cup \cdots \cup \Sigma_{j_m}) = \emptyset$, which implies

$$(\mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} \mid\mid \cdots \mid\mid \widehat{\mathcal{G}}_{jj_m}) \setminus \overline{\Sigma}_j \setminus \overline{\Sigma}_i \equiv (\mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} \mid\mid \cdots \mid\mid \widehat{\mathcal{G}}_{jj_m}) \setminus \overline{\Sigma}_i \tag{7}$$

Putting together (4)-(7), we obtain

$$\mathcal{M}_{j,i} \equiv (\mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} \mid\mid \cdots \mid\mid \widehat{\mathcal{G}}_{jj_m}) \setminus \overline{\Sigma}_i \tag{8}$$

By definition of $\widehat{\mathcal{G}}_{ij}$, we have

$$\widehat{\mathcal{G}}_{ij} \equiv \mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} \mid\mid \cdots \mid\mid \widehat{\mathcal{G}}_{jj_m} \tag{9}$$

which together with (8) yields

$$\mathcal{M}^{\mathcal{G}}_{j,i} \equiv \widehat{\mathcal{G}}_{ij} \setminus \overline{\Sigma}_i \tag{10}$$

as desired.

We can now prove correctness of Algorithm 1.

Theorem 1. *Let $\mathcal{G} = (V, E)$ with $V = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ be a tree-shaped interaction graph. The result of running Algorithm 1 on $\mathcal{G}$ are LTSs $\mathcal{K}_1, \ldots, \mathcal{K}_n$ such that $\mathcal{K}_i \equiv \widehat{\mathcal{G}} \setminus \overline{\Sigma}_i$ for every $\mathcal{L}_i \in V$.*

Proof. We have

$$\begin{aligned}
\mathcal{K}_i &\equiv \mathcal{L}_i \mid\mid (\mid\mid_{(\mathcal{L}_j,\mathcal{L}_i)\in E}\ \mathcal{M}^{\mathcal{G}}_{i,j}) && (11)\\
&\equiv \mathcal{L}_i \mid\mid (\mid\mid_{(\mathcal{L}_j,\mathcal{L}_i)\in E}\ (\widehat{\mathcal{G}}_{ij} \setminus \overline{\Sigma}_i)) && (12)\\
&\equiv \mathcal{L}_i \mid\mid (\mid\mid_{(\mathcal{L}_j,\mathcal{L}_i)\in E}\ \widehat{\mathcal{G}}_{ij}) \setminus \overline{\Sigma}_i && (13)\\
&\equiv \widehat{\mathcal{G}} \setminus \Sigma_i && (14)
\end{aligned}$$

Here, (11) follows from line 8 of the algorithm; (12) follows from Lemma 1; (13) follows from the fact that no two neighbours of $\mathcal{L}_i$ are connected by an edge, and so their sets of labels are disjoint. Finally, (14) follows from the definitions of $\widehat{\mathcal{G}}$ and $\widehat{\mathcal{G}}_{ij}$.

If the messages of Algorithm 1 are large in the worst case (in theory they can have the size of the full system) their number is optimal. More precisely, no MPA using fewer messages can be correct, where the only assumption we make about MPAs is that the output $\mathcal{K}_i$ is a function of $\mathcal{L}_i$ and the messages $\mathcal{L}_i$ receives from its neighbours.

Theorem 2. *For every correct MPA algorithm and every $n \geq 1$ there is a graph $\mathcal{G}_n$ such that the algorithm requires at least $2n - 2$ messages on $\mathcal{G}_n$.*

Proof. Assume there is a correct MPA $\mathcal{A}$ which always requires less than $2n - 2$ messages on graphs with n nodes. Consider the system $\mathcal{S}_1 = \mathcal{L}_1 \mid\mid \cdots \mid\mid \mathcal{L}_n$ where for every $1 \leq i \leq n$ the only maximal trace of $\mathcal{L}_i$ is $a_i a_{i+1} b_{i+1} b_i$ (its interaction graph is a line). Since the algorithm needs fewer than $2n - 2$ messages, there is an index i such that either $\mathcal{L}_i$ sends no message to $\mathcal{L}_{i-1}$, or sends no message to $\mathcal{L}_{i+1}$.

In the first case, consider the system $\mathcal{S}_2$ which is identical to $\mathcal{S}_1$, except that the only maximal trace of $\mathcal{L}_i$ is $a_i b_{i+1} a_{i+1} b_i$. For $\mathcal{S}_1$ the only maximal trace of $\mathcal{K}_n$ is $a_n a_{n+1} b_{n+1} b_n$, while for $\mathcal{S}_2$ the only maximal trace of $\mathcal{K}_n$ is $a_n a_{n+1}$. Since, by our assumption on MPAs, $\mathcal{L}_{i-1}$ does not receive any message from $\mathcal{L}_i$, it returns the same result $\mathcal{K}_n$ in both cases, and so the algorithm is incorrect.

In the second case, consider the system $\mathcal{S}_2$ which is identical to $\mathcal{S}_1$, except that the only maximal trace of $\mathcal{L}_{i+1}$ is $a_i b_{i+1} a_{i+1} b_i$, and proceed analogously with $\mathcal{K}_1$ instead of $\mathcal{K}_n$.

Observations. The restriction to tree-shaped interaction graphs can be weakened in several ways.

Communication Graphs. We can replace the interaction graph by a potentially smaller communication graph, thus reducing the number of messages.

Definition 7. *Let $\mathcal{G}$ be any subgraph of an interaction graph. An edge $(\mathcal{L}_i, \mathcal{L}_j) \in E$ is* redundant *if there exists a sequence $(\mathcal{L}_i, \mathcal{L}_{k_1})(\mathcal{L}_{k_1}, \mathcal{L}_{k_2}) \ldots (\mathcal{L}_{k_\ell}, \mathcal{L}_j)$ of edges such that $i \neq k_m \neq j$ and $\Sigma_{\mathcal{L}_{k_m}} \supseteq \Sigma_{\mathcal{L}_i} \cap \Sigma_{\mathcal{L}_j}$ for every $1 \leq k_m \neq j$.*

A communication graph of a system is any subgraph of the interaction graph obtained by iteratively removing redundant edges.

If some communication graph of a system is a tree then all its communication graphs are trees [1]. We then say that the system *lives on a tree.* The following proposition shows that the MPA can be applied to any system that lives on a tree, even if its interaction graph is not a tree.

Proposition 1. *Let $\mathcal{G} = (V, E)$ with $V = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ be a tree-shaped communication graph of $\mathcal{L} = \mathcal{L}_1 || \ldots || \mathcal{L}_n$. The result of running Algorithm 1 on $\mathcal{G}$ are LTSs $\mathcal{K}_1, \ldots, \mathcal{K}_n$ such that $\mathcal{K}_i \equiv \widehat{\mathcal{G}} \setminus \overline{\Sigma}_i$ for every $\mathcal{L}_i \in V$.*

Proof. The proof follows the lines of Theorem 2. We just have to adjust the arguments justifying Equations (6) and (7) in Lemma 1, and Equation (13) in Theorem 2. For (6) and (13) observe that, if the communication graph is a tree, then there are no edges between the neighbours of $\mathcal{L}_j$. Therefore, by the definition of communication graph, every common label of any two processes in (5) belongs to Σ_j which suffices to derive (6) and (13).

For (7) we observe that the communication graph also contains no edges between any of $\mathcal{L}_{j_1}, \ldots, \mathcal{L}_{j_m}$ and $\mathcal{L}_i$. So we have $(\Sigma_{j_1} \cup \ldots \cup \Sigma_{j_m}) \cap \mathcal{L}_i \subseteq \mathcal{L}_j$, and so

$$(\mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} || \cdots || \widehat{\mathcal{G}}_{jj_m}) \setminus \overline{\Sigma}_j \setminus \overline{\Sigma}_i \equiv (\mathcal{L}_j \mid\mid \widehat{\mathcal{G}}_{jj_1} || \cdots || \widehat{\mathcal{G}}_{jj_m}) \setminus \overline{\Sigma}_i$$

Tree decompositions. Any system $\mathcal{L} = \mathcal{L}_1 || \ldots || \mathcal{L}_n$ can be transformed into an equivalent one that lives on a tree. This is in itself trivial, since we can always choose this system as one single LTS equivalent to $\mathcal{L}$. However, this destroys the concurrency of the system. In order to preserve as much concurrency as possible we can compute a *tree decomposition* of a communication graph [18]. Every set $\{\mathcal{L}_{i_1}, \ldots, \mathcal{L}_{i_k}\}$ of the decomposition is then replaced by any single LTS equivalent to the subsystem $\mathcal{L}_{i_1}||\ldots||\mathcal{L}_{i_k}$. For instance, if the interaction graph of $\mathcal{L}_1||\ldots||\mathcal{L}_n$ is a ring with an even number of components, we can take the system $\mathcal{L}'_1||\ldots||\mathcal{L}'_{n/2}$, where $\mathcal{L}'_i \equiv \mathcal{L}_i||\mathcal{L}_{n-i+1}$ (Figure 4).

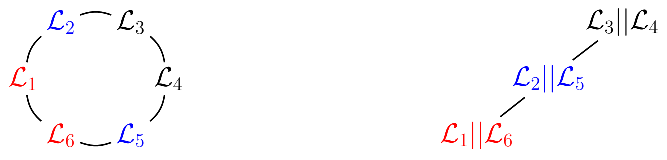

$\mathcal{L}_2$
$\mathcal{L}_3$
$\mathcal{L}_1$
$\mathcal{L}_4$
$\mathcal{L}_6$
$\mathcal{L}_5$
$\mathcal{L}_3||\mathcal{L}_4$
$\mathcal{L}_2||\mathcal{L}_5$
$\mathcal{L}_1||\mathcal{L}_6$

Fig. 4. A possible tree decomposition (right) of a ring-shaped interaction graph (left)

Computing one summary. Finally, we observe that computing one single update of component only requires to exchange $n-1$ messages instead of $2n-2$.

Proposition 2. *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs such that the system $\mathcal{L} = \mathcal{L}_1||\ldots||\mathcal{L}_n$ lives on a tree, and let $1 \leq i \leq n$. The update $\mathcal{K}_i$ of $\mathcal{L}_i$ can be computed by an MPA that uses only $n-1$ messages.*

Proof. Consider a communication graph $\mathcal{G} = (V, E)$ of $\mathcal{L} = \mathcal{L}_1, \ldots, \mathcal{L}_n$ that is a tree. If $n = 1$ then $i = 1$ and $\mathcal{L}_1 = \mathcal{K}_1$ is computed using $n - 1 = 0$ messages. Assume the proposition is true up to $n = k$. Consider $n = k + 1$ and take $1 \leq i \leq n$. Remark that $\mathcal{K}_i$ is computed exactly from the messages of the form $\mathcal{M}_{\mathcal{L}_j,\mathcal{L}_i}$ for $(\mathcal{L}_j, \mathcal{L}_i) \in E$ (denote by E_i the set containing these edges). Given such a $\mathcal{L}_j$, consider the largest subtree of $\mathcal{G}$ rooted in $\mathcal{L}_j$ which does not contains $\mathcal{L}_i$. This subtree has $n_j < k + 1$ nodes, so the update of $\mathcal{L}_j$ in this subtree can be computed from $n_j - 1$ messages. Remark that the hiding of $\overline{\Sigma}_i$ in this update is exactly $\mathcal{M}_{\mathcal{L}_j,\mathcal{L}_i}$. From that $\mathcal{K}_i$ is computed using $\sum_{(\mathcal{L}_j,\mathcal{L}_i)\in E_i}(n_j - 1) + |E_i| = n - 1$ messages.

3 Local Verification of Distributed Protocols on Trees

In this section we describe our implementations of Algorithm 1, tailored for checking local linear-time safety and liveness properties, respectively. Each implementation requires to

– Choose a congruence $\equiv$ that preserves the properties of interest.
– Resolve the nondeterminism introduced by $:\equiv$ in lines 4 and 8

Furthermore, $\mathcal{L}' :\equiv \mathcal{L}'$ should be implemented so that $\mathcal{L}'$ is as small as possible.

3.1 Safety

A local safety property of a component $\mathcal{L}_i$ within a system $(\mathcal{L}_1||\ldots||\mathcal{L}_n)$ is a property of the (observable) finite traces of $\mathcal{K}_i$, the update of $\mathcal{L}_i$. In order to preserve local safety properties we choose $\equiv$ as (observable) finite trace equivalence, i.e., $\mathcal{L} \equiv \mathcal{L}'$ if and only if $\mathcal{T}^*_{\mathcal{L}} = \mathcal{T}^*_{\mathcal{L}'}$. This equivalence is well known to be a congruence, and in the following we denote it by $\equiv_{\mathcal{T}}$. By Theorem 1, local safety properties of $\mathcal{L}_i$ can be decided by examining the traces of its update $\mathcal{K}_i$ returned by Algorithm 1.

We implement $\mathcal{L}' :\equiv_{\mathcal{T}} \mathcal{L}$ as follows: $\mathcal{L}'$ is the unique τ-free, minimal deterministic LTS equivalent to $\mathcal{L}$. More precisely,

$$\mathcal{L}' := MIN(DET(RED(\mathcal{L}))),$$

where RED, DET, MIN are algorithms for removing τ-transitions, determinizing, and minimizing LTSs, respectively. These algorithms are implemented using standard automata operations (see e.g. [19]).

This particular instantiation of Algorithm 1 closely corresponds to the MPA at the basis of the work presented in [2, 3].

3.2 Liveness

It is well known that defining a local liveness property of $\mathcal{L}_i$ as a property of the (observable) infinite traces of $\mathcal{K}_i$ is inadequate [20]. Consider two systems with sets of infinite traces $\{ab^\omega\}$ and $\{ab^\omega, ac^\omega\}$, respectively. If $\Sigma_i = \{a, b\}$, then in both cases the only infinite trace of $\mathcal{K}_i$ is ab^ω. However, in the first system $\mathcal{L}_i$ satisfies "after a eventually b", while in the second it does not. To solve this problem we keep information about the *divergences* of $\mathcal{L}_i$.

Definition 8. *Given $\mathcal{L} = (\Sigma, S, T, s^0)$ with transitions labelled by τ, a* divergence *of $\mathcal{L}$ is a finite observable trace tr of $\mathcal{L}$ such that there exists an infinite trace tr' of $\mathcal{L}$ verifying $tr'|_\Sigma = tr$. The set of divergences of $\mathcal{L}$ is denoted by $\mathcal{D}_{\mathcal{L}}$.*

Given $(\mathcal{L}_1||\ldots||\mathcal{L}_n)$ without τ-transitions, a divergence *of $\mathcal{L}_i$ in $\mathcal{L}$ is a finite observable trace $tr_i \in \mathcal{T}^*_{\mathcal{L}_i}$ such that there exists an infinite observable trace $tr \in \mathcal{T}^\omega_{\mathcal{L}}$ satisfying $tr|_{\Sigma_i} = tr_i$.*

The hiding operation links these two definitions: Given $\mathcal{L} = \mathcal{L}_1||\ldots||\mathcal{L}_n$ without τ-transitions, the set of divergences of $\mathcal{L}_i$ in $\mathcal{L}$ is equal to the set $\mathcal{D}_{\mathcal{L}\setminus\overline{\Sigma_i}}$ of divergences of $\mathcal{L} \setminus \overline{\Sigma_i}$.

We define a local liveness property of $\mathcal{L}_i$ as a property of $\mathcal{T}_{\mathcal{K}_i}$ and $\mathcal{D}_{\mathcal{L}\setminus\overline{\Sigma_i}}$, and so we choose: $\mathcal{L} \equiv \mathcal{L}'$ if and only if $\mathcal{T}_{\mathcal{L}} = \mathcal{T}_{\mathcal{L}'}$ and $\mathcal{D}_{\mathcal{L}} = \mathcal{D}_{\mathcal{L}'}$. This equivalence is known to be a congruence for LTSs (see [21] for example). In the following we denote it by $\equiv_{\mathcal{D}}$.

By Theorem 1, local liveness properties of $\mathcal{L}_i$ can be decided by examining the traces of the updated version $\mathcal{K}_i$ obtained by Algorithm 1.

In order to implement $\mathcal{L}' :\equiv_{\mathcal{D}} \mathcal{L}$, we profit from the following fact: for finite LTSs, $\mathcal{T}_{\mathcal{L}} = \mathcal{T}_{\mathcal{L}'}$ iff $\mathcal{T}^*_{\mathcal{L}} = \mathcal{T}^*_{\mathcal{L}'}$, and so $\mathcal{L} \equiv_{\mathcal{D}} \mathcal{L}$ iff $\mathcal{T}^*_{\mathcal{L}} = \mathcal{T}^*_{\mathcal{L}}$ and $\mathcal{D}_{\mathcal{L}} = \mathcal{D}_{\mathcal{L}'}$. This allows us to replace the nondeterministic assignment by a five-step procedure:

$$\mathcal{L}' := HID(MIN(DET(RED(DIV(\mathcal{L}))))),$$

where MIN, DET, and RED are defined as above. For $\mathcal{L} = (\Sigma, S, T, s^0)$, $DIV(\mathcal{L})$ is the LTS $(\Sigma_d, S_d, T_d, s^0_d)$ such that: $\Sigma_d = \Sigma \cup \{\tau'\}$ with $\tau' \notin \Sigma$, $S_d = S$, $T_d = T \cup T_d^{\tau'}$ with $T_d^{\tau'} = \{(s,s) \ : \ \exists s', s \overset{\tau}{\rightsquigarrow}_{\mathcal{L}} s' \overset{\tau}{\rightsquigarrow}_{\mathcal{L}} s'\}$, and $s^0_d = s^0$. Finally, $HID(\mathcal{L})$ is defined as $\mathcal{L} \setminus \{\tau'\}$, where τ' has to be the same as used in DIV. We have:

Theorem 3. *$\mathcal{L} \equiv_{\mathcal{D}} HID(MIN(DET(RED(DIV(\mathcal{L})))))$ for any LTS $\mathcal{L}$.*

Proof. Denote by Σ the set of labels of $\mathcal{L}$. First remark that a finite sequence tr of labels from Σ is a divergence of $\mathcal{L}$ if and only if $tr\tau'^{\omega}$ is an infinite observable trace of $DIV(\mathcal{L})$. This is because (1) tr is a divergence of $\mathcal{L}$ if and only if there exists a path realizing tr in $\mathcal{L}$ and reaching a state from which there exists an infinite path using only τ-transitions, and (2) $DIV(\mathcal{L})$ is an exact copy of $\mathcal{L}$ with the addition of τ'-transitions, all of the form (s,s) for $s \in S_d = S$, and there is a τ'-transition from a state s to itself if and only if there exists in $\mathcal{L}$ an infinite path using only τ-transitions and starting from s.

From this, and because MIN, DET, and RED preserve observable traces, one gets that a finite sequence tr of labels from Σ is a divergence of $\mathcal{L}$ if and only if $tr\tau'^{\omega}$ is an infinite observable trace of $MIN(DET(RED(DIV(\mathcal{L}))))$. Also remark that $MIN(DET(RED(DIV(\mathcal{L}))))$ does not contain any τ-transition.

The remark that HID only replaces τ'-transitions by τ-transitions then allows to conclude that a finite sequence tr of labels from Σ is a divergence of $\mathcal{L}$ if and only if $tr\tau^{\omega}$ is an infinite trace of $HID(MIN(DET(RED(DIV(\mathcal{L})))))$ if and only if tr is a divergence of $HID(MIN(DET(RED(DIV(\mathcal{L})))))$.

4 Experimental Evaluation

The approaches described in the previous section have been implemented as an extension of the planner DISTOPLAN [3]. In this section we report on an experimental evaluation of the performances of this implementation on two protocols: a mutual exclusion algorithm on trees [4] and the pragmatic general multicast protocol [5]. All experiments were performed using the same computer with an Intel Core i5 processor and a memory limit set to 4GB.

4.1 Raymond's Mutual Exclusion Protocol

In [4], Raymond presents a distributed protocol ensuring mutual exclusion for n processes organized as a tree. Processes communicate by rendez-vous, which allows us to model the protocol as a parallel composition of LTSs, one for each process. The unique communication graph is given by the tree. The scaling parameter is the number of processes, which fits well with our approach as the difference between two instances of the protocol is due to the number of LTSs needed to model them, rather than to their sizes.

The protocol can be roughly described as follows. A single token is passed between the processes, a process being allowed to access its critical section only if it owns the token. At any time, each process P_i not holding the token knows which of its neighbours in the tree is closest to the token. In other words P_i knows in which maximal subtree containing exactly one of its neighbours, but not P_i itself, the token currently is. Its requests for the token (and all requests by other processes that it may have to transmit) are sent to this particular neighbour. A more precise description of this protocol is given in Algorithm 2. It describes in a Promela-like manner an agent called x. $\mathcal{N}$ denotes the set of neighbours of x in the tree of agents on which the protocol is executed. We consider that x $\in \mathcal{N}$. $\mathcal{Q}(\mathcal{N})$ denotes the set of queues of elements from $\mathcal{N}$. Notice that the same element never appears twice in the queue requestQ. At each step one of three mutually exclusive guarded atomic instruction sequences is executed: (1) re-assignation of the token when x holds it, (2) request for the token, (3) non-deterministic choice between: asking for entering the critical section, receiving a request message from a neighbour, receiving the token from a neighbour, deciding to exit the critical section.

Algorithm 2. An agent for Raymond's mutual exclusion protocol

```
Agent x (holder: $\mathcal{N}$, using: $\{true, false\}$, requestQ: $\mathcal{Q}(\mathcal{N})$, asked: $\{true, false\}$)
holder=x $\wedge$ using=$false$
        $\wedge$ isnotempty(requestQ) $\rightarrow$ holder:=dequeue(requestQ)
                                      asked:=$false$
                                      holder=x $\rightarrow$ using:=$true$
                                      holder$\neq$x $\rightarrow$ !token to holder
holder$\neq$x $\wedge$ asked=$false$
        $\wedge$ isnotempty(requestQ) $\rightarrow$ !request(x) to holder
                                      asked:=$true$
else $\rightarrow$ isnotin(x,requestQ) $\rightarrow$ enqueue(x,requestQ)
        ?request(n) $\rightarrow$ enqueue(n,requestQ)
        ?token $\rightarrow$ holder:=x
        using=$true$ $\rightarrow$ using:=$false$
```

We consider instances of this protocol in which processes form a complete binary tree. The results obtained are presented in Table 1. The leftmost columns give the depth of the binary tree considered (Depth) and the corresponding number of processes (Processes). For each depth the column **Traces** reports the time (in seconds) needed for the following tasks: (1) run Algorithm 1 to obtain the updated versions of all the processes with respect to trace equivalence $\equiv_{\mathcal{T}}$ (subcolumn MPA); (2) same but computing only $n-1$ messages in order to obtain the updated version of the root only (subcolumn OneWay); (3) verify a simple local property from the updated root (subcolumn Verification). Column

Table 1. Results for the analysis of Raymond's mutual exclusion protocol on complete binary trees. Times are in seconds.

Depth	Processes	Traces			Divergences		
		MPA	**OneWay**	**Verification**	**MPA**	**OneWay**	**Verification**
2	3	0.12	0.15	<0.01	0.14	0.13	<0.01
3	7	1.41	1.23	<0.01	2.07	1.88	<0.01
4	15	2.36	2.20	<0.01	4.58	4.36	<0.01
5	31	5.29	4.67	<0.01	10.44	9.67	<0.01
6	63	10.62	9.63	<0.01	21.81	20.27	<0.01
7	127	21.94	19.70	<0.01	44.86	41.55	<0.01

Divergences gives the times required by analogous tasks, but with respect to divergence equivalence $\equiv_{\mathcal{D}}$.

We check a safety (in the case of $\equiv_{\mathcal{T}}$) and a liveness (in the case of $\equiv_{\mathcal{D}}$) property in order to compare our MPA with an algorithm that constructs the state space, for which we use Spin. The local safety property verified for the root processes is: "It is not possible to request the token twice without receiving it in between". Remark that, it is also expressible as the following global property: "It is not possible for the root process to request the token twice without receiving it in between". We verify this property by checking emptiness of the product of the automaton φ on the left of Figure 5 (representing the negation of the property) and the (projection onto the labels of φ of the) updated version of the root process obtained by Algorithm 1 (in which all states are considered accepting). The local liveness property verified for the root processes is: "The token is received in finite time after any request". Similarly, the property is verified by checking emptiness of the product of the Büchi automaton φ' on the right of Figure 5 (representing the negation of the property) with the (projection on the labels of φ' of the) updated version of the root process obtained by Algorithm 1 (in which all states are considered accepting).

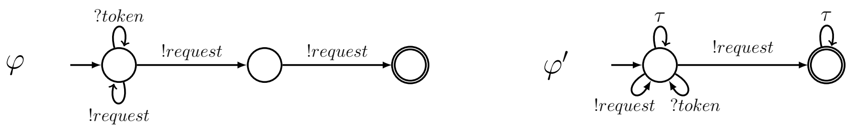


Fig. 5. Two properties to be checked on Raymond's mutual exclusion protocol. *!request* is a general token request action and *?token* is a general token reception action.

Analysis of the Results. As expected, the approach scales very well with the number of components: at each depth the number of components almost doubles while the time spent for computing the updates of the components is slightly more than doubling. We compare it with a verification of the same properties using Spin [22]. Since the system is highly concurrent, we use Spin with the partial order reduction optimization. Spin outperforms our approach for the complete binary tree of depth 2 (it needs less than 0.01 seconds to verify the properties). For trees of depth 3 or greater, however, Spin runs out of memory (memory limit being 4GB). Notice that, since the properties we check are true, Spin needs a full state-space exploration to verify them, while our MPA prevents this. We also remark that the time needed by our MPA is almost entirely spent in computing the updates of the components, and so the additional cost of verifying other properties after the first one is small, since the previously computed updates can be re-used. Finally, observe that, in this example, the difference in time spent between the standard application of Algorithm 1 (computing all updates) and the case where only the update of the root is computed (so only half of the messages are constructed) is not significant. This can be explained by the fact that – being the initial owner of the token – the root imposes more constraints to the system than the other components. So the messages from the root are potentially much simpler to compute than the messages to the root.

4.2 The Pragmatic General Multicast Protocol

[5] describes the pragmatic general multicast protocol (PGM), a reliable distributed protocol for distributing information from multiple senders to multiple receivers in a network, designed to minimize the load of the network due to acknowledgement messages and retransmissions of lost messages. We consider the specific version of this very generic protocol given in Algorithm 3 (which is almost the one described in [17]): a unique source sends information to multiple receivers in a network organized as a tree. Each process is described in a Promela-like manner and consists of a single loop in which a non-deterministic choice is done at each step between several guarded atomic instruction sequences.

The source can receive a negative acknowledgement $nak(nr)$ for some data, in this case it sends back a confirmation $ncf(nr)$ and, if the data nr is still within range of its window it adds it to the set $recNak$ of data to be re-sent. If some data nr is in the set $recNak$ and still within range of the window this data can be re-sent as a message $rdata(nr, txWTr)$. And, if no data needs to be re-sent, a new data can be sent as a message $odata(nr, txWTr)$ and the window may be moved. Any network element can receive negative acknowledgements and propagate them above itself in the tree. It can also transmit data below itself in the tree. Finally, it may generate new negative acknowledgements while no confirmation have been received for them. A receiver can receive data, and, when it allows it to deduce that some data are missing (by looking at the previously received data and because of the fact that the data are consecutive integers) it can send negative acknowledgements for these data.

As before we represent each process by an LTS. However, communications are no longer by rendez-vous but use messages sent through bidirectional channels (which can lose messages). Each channel is thus also modelled as an LTS. The unique communication graph of such a system is a tree.

In our experiments we considered two possible topologies for the systems: lines and complete binary trees. In each of these cases we considered instances of the protocol with increasing numbers of processes. We also made other parameters

Algorithm 3. PGM: source, network elements, and receivers

```
Source (data: ℕ, winSize: ℕ, txWTr: ℕ, recNak: $2^{\mathbb{N}}$)
?nak(nr) → !ncf(nr)
               txWTr < nr → recNak := add(recNak,nr)
isin(nr,recNak) → nr > txWTr → !rdata(nr,txWTr)
                      recNak := remove(recNak,nr)
lenght(recNak) = 0 → !odata(data,txWTr)
                           data := data + 1
                           odata > winSize + txWTr → txWTr := txWTr + winSize

Network element (setRepair: $2^{\mathbb{N}}$)
?nak(nr) → setRepair := add(setRepair,nr)
               isnotin(setRepair,nr) → !nak(nr) upwards
               !ncf(nr) downwards
?rdata(nr,txWTr) → setRepair := remove(setRepair,nr)
                         !rdata(nr,txWTr) downwards
?ncf(nr) → setRepair := remove(setRepair,nr)
?odata(nr,txWTr) → !odata(nr,txWTr) downwards
isin(nr,setRepair) → !nak(nr) upwards

Receiver (rxWTr: ℕ, setNr: $2^{\mathbb{N}}$, setMissing: $2^{\mathbb{N}}$)
?odata(nr,txWTr) ∧ rxWTr < nr → rxWTr < txWTr → rxWTr := txWTr
                                       setNr := add(setNr,nr)
                                       for all (rxWTr < i < nr ∧ isnotin(i,setNr))
                                           setMissing := add(setMissing,i)
                                       setMissing := remove(setMissing,nr)
?rdata(nr,txWTr) ∧ rxWTr < nr → rxWTr < txWTr → rxWTr := txWTr
                                       setNr := add(setNr,nr)
                                       setMissing := remove(setMissing,nr)
isin(nr,setMissing) → nr > rxWTr → !nak(nr)
                         nr ≤ rxWTr setMissing := remove(setMissing,nr)
```

vary: the number of different data to be sent by the source (two or three) and the capacity of the channels (one or two messages). The initial value of *data* for the source is set to 1 and the value of *winSize* is set to anything higher than the number of different data to be sent. All other integer parameters are initialized to 0 and all sets are initially empty.

Figure 6 presents the update of a leaf as obtained by Algorithm 1 (using $\equiv_{\mathcal{T}}$ as congruence) when ran on a complete binary tree of depth five with two data to be sent and channels of capacity one. It is interesting to notice that just by looking at this LTS, a behaviour (corresponding to the path with larger labels in the figure) that may not be directly anticipated from the description of the protocol can be remarked: it is possible for the leaf to deduce that the second (and last) data will never be received. This can in fact be explained by the possible losses of messages. For sure $ncf(1)$ has been sent (by the source after reception of $nak(1)$ from another leaf) after $odata(2, 0)$. So, each channel between the source and the leaf represented in Figure 6 has contained $ncf(1)$ at some time, and if it has contained $odata(2,0)$ at some time it was before $ncf(1)$. The only explanation for receiving $ncf(1)$ before $odata(2,0)$ is thus a loss of $odata(2,0)$ at some channel.

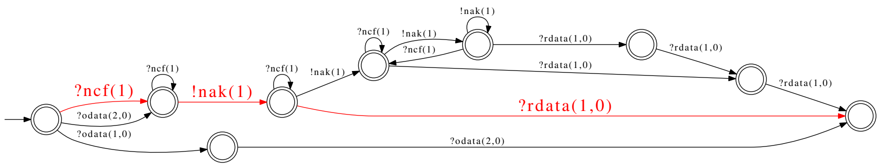


Fig. 6. PGM protocol: update of a leaf in a complete binary tree of depth five when the number of different data to be sent is two and the channels capacity is one

Table 2 gives the results obtained for the PGM protocols on lines, in the case where the source can only send two different data and the channels are of capacity one. Results are organized as before. The only difference are in the Basic and MPA columns. The Basic column presents the times obtained by running Algorithm 1 while the MPA columns present the times obtained by running Algorithm 4. This algorithm is a variation of Algorithm 1 where messages that "cross" at some edge of the communication graph are not completely independent: the constraints imposed by the first to be computed are used to compute the second.

Algorithm 4. Variation of Algorithm 1

```
Input: an interaction graph $\mathcal{G} = (V, E)$ with $V = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$
 1: $\mathsf{M} \leftarrow E$
 2: while $\mathsf{M} \neq \emptyset$ do
 3:     choose $(\mathcal{L}_i, \mathcal{L}_j) \in \mathsf{M}$ such that $(\mathcal{L}_k, \mathcal{L}_i) \notin \mathsf{M}$ for every $k \neq j$
 4:     $\mathcal{L}_i :\equiv \mathcal{L}_i \mid\mid (\mid\mid_{k \neq j, (\mathcal{L}_k, \mathcal{L}_i) \in E} \mathcal{M}_{k,i})$
 5:     $\mathcal{M}_{i,j} :\equiv \mathcal{L}_i \setminus \overline{\Sigma}_j$
 6:     remove $(\mathcal{L}_i, \mathcal{L}_j)$ from $\mathsf{M}$
 7: end while
 8: for all $i \in V$ do
 9:     $\mathcal{K}_i :\equiv \mathcal{L}_i \mid\mid (\mid\mid_{(\mathcal{L}_j, \mathcal{L}_i) \in E} \mathcal{M}_{j,i})$
10: end for
```

The local safety and liveness properties we check at the source are the following: "The last data can only be sent once" (its negation is represented by the FSA φ on the left of Figure 7) and "The first data is always sent at least once" (its negation is represented by the NBA φ' on the right of Figure 7). The verification process is the same as above.

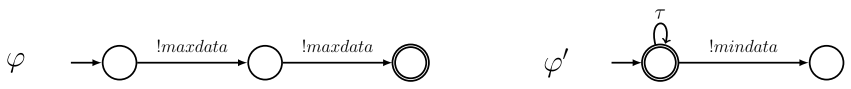


Fig. 7. Two properties to be checked on PGM. !*maxdata* (respectively !*mindata*) represents any sending of the last data (respectively the first data) using an odata or an rdata message.

Table 2. Analysis of PGM (with two different data and channels of capacity one) on lines. Times are in seconds.

Processes	**Traces**				**Divergences**		
	Basic	**MPA**	**OneWay**	**Verification**	**MPA**	**OneWay**	**Verification**
5	7.79	0.08	0.03	<0.01	0.11	0.08	<0.01
10	20.27	0.13	0.08	<0.01	0.16	0.13	<0.01
15	32.76	0.19	0.15	<0.01	0.22	0.20	<0.01
20	41.99	0.23	0.16	<0.01	0.26	0.20	<0.01
25	53.14	0.26	0.21	<0.01	0.31	0.24	<0.01
30	67.50	0.30	0.25	<0.01	0.37	0.27	<0.01
35	77.32	0.35	0.29	<0.01	0.43	0.34	<0.01
40	89.95	0.40	0.32	<0.01	0.49	0.36	<0.01
45	101.25	0.46	0.36	<0.01	0.57	0.40	<0.01
50	113.60	0.50	0.40	<0.01	0.60	0.44	<0.01

Table 3. Analysis of PGM on lines using Algorithm 4. Different numbers of data to be sent (d) and different sizes of channels (c) are considered. Times are in seconds.

Processes	**Traces**		**Divergences**	
	d=2, c=2	**d=3, c=1**	**d=2, c=2**	**d=3, c=1**
5	10.71	10.63	15.37	13.26
10	19.19	12.60	28.94	18.00
15	27.24	14.56	41.77	22.21
20	35.53	16.46	55.77	26.80
25	43.66	18.24	68.40	30.95
30	52.14	20.66	81.43	35.36
35	60.16	22.64	95.39	39.82
40	68.78	24.80	109.17	44.49
45	77.00	26.66	122.57	48.56
50	85.12	29.01	136.60	53.27

Table 3 gives the other results obtained for the PGM protocols on lines. Table 4 gives the results obtained for the PGM protocols on complete binary trees. They only report the running times of Algorithm 4 as the verification of the properties considered still always requires less than 0.01 seconds.

Table 4. Analysis of PGM on complete binary trees using Algorithm 4. Different numbers of data to be sent (d) and different sizes of channels (c) are considered. Times are in seconds.

Depth	Proc.	Traces			Divergences		
		d=2, c=1	**d=2, c=2**	**d=3, c=1**	**d=2, c=1**	**d=2, c=2**	**d=3, c=1**
3	7	0.85	26.26	59.30	1.41	33.96	93.02
4	15	1.58	56.10	114.05	1.60	72.89	156.93
5	31	2.48	113.82	235.32	2.93	153.47	316.63
6	63	5.06	231.27	472.19	5.73	310.28	641.13
7	127	10.24	474.57	979.85	12.10	625.61	1582.23

Analysis of the Results. Spin can deal with lines of length 5 within 20 seconds but cannot handle larger lines nor binary trees without running out of the 4GB of memory allowed to it. In addition to what has been noticed in the case of Raymond's mutual exclusion protocol, it appears that using Algorithm 4 instead of Algorithm 1 can significantly reduce running times. This is due to the fact that some constraints are taken into account earlier, and so some messages can be simplified. Using this version of our approach is not always that efficient however. In the case of Raymond's mutual exclusion protocol, for example, it almost does not reduce running times. The comparison of the different numbers of data and capacities of channels also shows that, if our approach scales well with the number of components of the system to analyse, it is more sensitive to increases of the sizes of these components.

Conclusion

We have presented message-passing algorithms for the verification of local properties of distributed protocols. The components of the protocol must have a tree-shaped communication structure. The MPAs compute for each component $\mathcal{L}_i$ an LTS equivalent to the result of hiding in the full LTS (the LTS of the full protocol) all actions not appearing in $\mathcal{L}_i$. We have shown that the MPAs can be instantiated with different equivalence notions, in particular trace equivalence and divergence equivalence. We have evaluated the algorithms on two well-known protocols, and shown that for several important properties they scale very well, in particular much better than a generic search-based model-checker like Spin.

The properties we have used for our comparison with Spin were true properties, i.e., properties that hold for the protocol. In fact, for false properties Spin often outperforms our approach, possibly due to the existence of a relatively large number of counterexamples, which allows Spin to quickly find one. This suggests to run Spin and a suitable MPA in parallel.

Future work will explore how to instantiate the MPAs with equivalence relations sensitive to deadlocks or partial deadlocks.

References

1. Fabre, E.: Bayesian Networks of Dynamic Systems. Habilitation à diriger des recherches, Université de Rennes1 (2007)
2. Fabre, E., Jezequel, L.: Distributed optimal planning: an approach by weighted automata calculus. In: CDC, pp. 211–216 (2009)
3. Fabre, E., Jezequel, L., Haslum, P., Thiébaux, S.: Cost-optimal factored planning: Promises and pitfalls. In: ICAPS, pp. 65–72 (2010)
4. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. TCS 7(1), 61–77 (1989)
5. Speakman, T., et al.: PGM reliable transport protocol specification. RFC 3208 (Experimental) of the IETF (2001)
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. 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)
8. Graf, S., Steffen, B.: Compositional minimization of finite state systems. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 186–196. Springer, Heidelberg (1991)
9. Grumberg, O., Long, D.E.: Model checking and modular verification. TOPLAS 16(3), 843–871 (1994)
10. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hullance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)
11. FRD2 user manual (2009)
12. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: A semantics-based tool for the verification of concurrent systems. TOPLAS 15(1), 36–72 (1993)
13. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT 15(2), 89–107 (2013)
14. Abdulla, P.A.: Regular model checking. STTT 14(2), 109–118 (2012)
15. Bérard, B., Bouyer, P., Petit, A.: Analysing the PGM protocol with UPPAAL. International Journal of Production Research 42(14), 2773–2791 (2004)
16. Boyer, M., Sighireanu, M.: Synthesis and verification of constraints in the PGM protocol. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 264–281. Springer, Heidelberg (2003)
17. Esparza, J., Maidl, M.: Simple representative instantiations for multicast protocols. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 128–143. Springer, Heidelberg (2003)
18. Bodlaender, H.: A linear time algorithm for finding tree-decompositions of small treewidth. In: STC, pp. 226–234 (1993)
19. Sakarovitch, J.: Éléments de théorie des automates. Vuibert (2003)
20. Brookes, S.D., Roscoe, A.W.: An improved failures model for communicating processes. In: Brookes, S.D., Winskel, G., Roscoe, A.W. (eds.) Seminar on Concurrency. LNCS, vol. 197, pp. 281–305. Springer, Heidelberg (1985)
21. Valmari, A.: All linear-time congruences for finite LTSs and familiar operators. In: ACSD (2012)
22. Holzmann, G.: The SPIN model checker: primer and reference manual. Addison-Wesley Professional (2003)

Safety Problems Are NP-complete for Flat Integer Programs with Octagonal Loops

Marius Bozga[1], Radu Iosif[1], and Filip Konečný[2]

[1] VERIMAG/CNRS, Grenoble, France
[2] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. This paper proves the NP-completeness of the reachability problem for the class of flat counter machines with difference bounds and, more generally, octagonal relations, labeling the transitions on the loops. The proof is based on the fact that the sequence of powers $\{R^i\}_{i=1}^{\infty}$ of such relations can be encoded as a periodic sequence of matrices, and that both the prefix and the period of this sequence are $2^{O(\|R\|_2)}$ in the size of the binary encoding $\|R\|_2$ of a relation R. This result allows to characterize the complexity of the reachability problem for one of the most studied class of counter machines [6,10], and has a potential impact on other problems in program verification.

1 Introduction

Counter machines are powerful abstractions of programs, commonly used in software verification. Due to their expressive power, counter machines can simulate Turing machines [18], hence, in theory, any program can be viewed as a counter machine. In practice, effective reductions to counter systems have been designed for programs with dynamic heap data structures [3], arrays [5], dynamic thread creation and shared memory [1], etc. Since counter machines with only two variables are Turing-complete [18], all their decision problems (reachability, termination) are undecidable. This early negative result motivated researchers to find classes of systems with decidable problems, such as: (branching) vector addition systems [13,17], reversal-bounded counter machines [16], Datalog programs with gap-order constraints [20], and flat counter machines [2,10,6]. Despite the fact that reachability of a set of configurations is decidable for these classes, few of them are actually supported by tools, and used for real-life verification purposes. The main reason is that the complexities of the reachability problems for these systems are, in general, prohibitive. Thus, most software verifiers rely on incomplete algorithms, which, due to the loss of precision, may raise large numbers of false alarms. Improving the precision of these tools requires mixed techniques such as combinations of *static analysis* and *acceleration* and relies on identifying subproblems for which the set of reachable states, or the transitive closure of the transition relation, can be computed precisely [14].

We study the complexity of the reachability problems for a class of *flat counter machines* (i.e., the control structure forbids nested loops), in which the transitions occurring inside loops are all labeled with *difference bounds constraints*, i.e. conjunctions of linear inequalities of the form $x - y \leq c$ where $x, y \in \mathbf{x} \cup \mathbf{x}'$ and $c \in \mathbb{Z}$ is a constant.

Furthermore, we extend the result to the case of octagonal relations, which are conjunctions of the form $\pm x \pm y \leq c$.

The decidability of the reachability problem for these classes relies on the fact that the transitive closures R^+ of relations R, defined by difference bounds and octagonal constraints, are expressible in Presburger arithmetic [10]. In [6], we presented a concise proof of this fact, based on the observation that any sequence of powers $\{R^i\}_{i=1}$, can be encoded as a *periodic sequence* of matrices, which can be defined by a quantifier-free Presburger formula whose size depends on the prefix and the period of the matrix sequence. In this paper we show primarily that both the prefix and period and this sequence are of the order of $2^{O(\|R\|_2)}$, where $\|R\|_2$ is the size of the binary encoding of the relation. More precisely, the quantifier-free Presburger formula defining a transitive closure (and, implicitly, the reachability problem for the counter machine) has $2^{O(\|R\|_2)}$ many disjuncts of polynomial size. A non-deterministic Turing machine that solves the reachability problem can guess, for each loop relation R, the needed disjunct of R^+, and validate its guess in NPTIME($\|R\|_2$).

Related Work. The complexity of safety, and, more generally, temporal logic properties of integer counter machines has received relatively little attention. For instance, the exact complexity of reachability for vector addition systems (VAS) is an open problem (the only known upper bound is non-primitive recursive), while the coverage and boundedness problems are EXPSPACE-complete for VAS [19], and 2EXPTIME-complete for branching VAS [13].

In [15] the authors study the functional equivalence of programs with increment, decrement and zero test, in the *reversal-bounded* case, where the counters are allowed to switch between non-decreasing and non-increasing modes a number of times which is bounded by a constant. It is found that the equivalence problem is in PSPACE, while the in-equivalence problem is NP-complete. Our model of computation is incomparable, since flat programs with non-deterministic updates are not reversal-bounded.

On what concerns counter machines with gap-order constraints (a restriction of difference bounds constraints $x-y \leq c$ to the case $c \leq 0$), reachability is PSPACE-complete [9], even in the absence of the flatness restriction on the control structure. Our result is incomparable to [9], as we show NP-completeness for flat counter machines with more general[1], difference bounds relations on loops.

The results which are probably closest to ours are the ones in [12,11], where flat counter machines with deterministic transitions of the form $\bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ji} \cdot x_i + b_{ji} \leq 0 \wedge \bigwedge_{i=1}^{n} x_i' = x_i + c_i$ are considered. In [12] it is shown that model-checking LTL is NP-complete for these systems, matching thus our complexity for reachability with difference bounds constraints, while model-checking first-order logic and linear μ-calculus is PSPACE-complete [11], matching the complexity of CTL* model checking for gap-order constraints [9]. These results are again incomparable with ours, since (i) the linear guards are more general, while (ii) the vector addition updates are more restrictive (e.g. the direct transfer of values $x_i' = x_j$ for $i \neq j$ is not allowed).

[1] The generalization of gap-order to difference bound constraints suffices to show undecidability of non-flat counter machines, hence the restriction to flat control structures is crucial.

2 Preliminary Definitions

We denote by $\mathbb{Z}$ and $\mathbb{N}$ the sets of integers and positive integers, and let $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$. We write $[n]$ for the interval $\{0,\ldots,n-1\}$, $\mathrm{abs}(n)$ for the absolute value of the integer $n \in \mathbb{Z}$, and $lcm(n_1,\ldots,n_k)$ for the least common multiple of $n_1,\ldots,n_k \in \mathbb{N}$. Let $\mathbf{x}$ denote a nonempty set of variables, and $\mathbf{x}' = \{x' \mid x \in \mathbf{x}\}$. A *valuation* of $\mathbf{x}$ is a function $\nu : \mathbf{x} \to \mathbb{Z}$. The set of all such valuations is denoted by $\mathbb{Z}^{\mathbf{x}}$, and we denote by $\mathbb{Z}^N$ the N-times cartesian product $\mathbb{Z} \times \ldots \times \mathbb{Z}$, for some $N > 0$. We assume that the reader is familiar with Presburger arithmetic, and we denote by QFPA (quantifier-free Presburger arithmetic) the set of boolean combinations of linear inequalities and linear modulo constraints. For a QFPA formula ϕ, let $Atom(\phi)$ denote the set of atomic propositions in ϕ, and $\varphi[t/x]$ denote the formula obtained by substituting the variable x with the term t in φ.

A formula $\phi(\mathbf{x},\mathbf{x}')$ is evaluated with respect to two valuations $\nu_1,\nu_2 \in \mathbb{Z}^{\mathbf{x}}$, by replacing each occurrence of $x \in \mathbf{x}$ with $\nu_1(x)$ and each occurrence of $x' \in \mathbf{x}'$ with $\nu_2(x)$ in ϕ. The satisfaction relation is denoted by $(\nu_1,\nu_2) \models \phi(\mathbf{x},\mathbf{x}')$. A formula $\phi_R(\mathbf{x},\mathbf{x}')$ is said to *define* a relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ whenever for all $\nu_1,\nu_2 \in \mathbb{Z}^{\mathbf{x}}$, $(\nu_1,\nu_2) \in R$ if and only if $(\nu_1,\nu_2) \models \phi_R$. The composition of two relations $R_1,R_2 \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ defined by formulae $\varphi_1(\mathbf{x},\mathbf{x}')$ and $\varphi_2(\mathbf{x},\mathbf{x}')$, respectively, is the relation $R_1 \circ R_2$, defined by the formula $\exists \mathbf{y} \,.\, \varphi_1(\mathbf{x},\mathbf{y}) \wedge \varphi_2(\mathbf{y},\mathbf{x}')$. The *identity relation* $Id_{\mathbf{x}}$ is defined by the formula $\bigwedge_{x\in\mathbf{x}} x' = x$.

Definition 1. *A* class of relations *is a set $\mathcal{R}$ of QFPA formulae $\phi_R(\mathbf{x},\mathbf{x}')$ defining relations $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$, such that $Id_{\mathbf{x}}$ is $\mathcal{R}$-definable, and, for any two $\mathcal{R}$-definable relations $R_1,R_2 \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$, their composition $R_1 \circ R_2$ is $\mathcal{R}$-definable.*

Notice that any set $\mathcal{R}$ of formulae $\varphi(\mathbf{x},\mathbf{x}')$ that has quantifier elimination is a class of relations. If the class of a relation is not specified a priori, we consider it to be the set of all QFPA formulae. Given a relation R, we denote by R^i, for $i > 0$, the i-times composition of R with itself, and by R^0 the identity relation $Id_{\mathbf{x}}$. We denote by $R^+ = \bigcup_{i=1}^{\infty} R^i$ the *transitive closure* of R. Notice that, if R is an $\mathcal{R}$-definable relation, then the sequence $\{R^i\}_{i\geq 0}$ is $\mathcal{R}$-definable as well. In the following, we sometimes use the same symbol to denote a relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ and the formula $\phi_R(\mathbf{x},\mathbf{x}')$ defining it.

For a constant $c \in \mathbb{Z}$, we denote by $\|c\|_2 = \lceil \log_2(\mathrm{abs}(c)) \rceil$, if $\mathrm{abs}(c) > 2$ and $\|c\|_2 = 2$, otherwise, the *size of its binary encoding*[2]. The *binary size* of a formula is the sum of the binary sizes of its coefficients. It is known that the satisfiability problem for QFPA is NP-complete in the binary size of the formula [22]. The binary size of an $\mathcal{R}$-definable[3] relation R is $\|R\|_2^{\mathcal{R}} = \min\{\|\phi_R\|_2 \mid \phi_R \in \mathcal{R},\ \phi_R \text{ defines } R\}$. When the class of a relation is obvious from the context, it will be omitted. For space reasons, all proofs and missing material are given in [7].

[2] Abstracting from particular machine representations, we assume that at least 2 bits are needed to encode each integer.

[3] The class $\mathcal{R}$ is relevant here, because the same relation can be defined by a smaller formula not in $\mathcal{R}$

3 The Reachability Problem for Flat Counter Machines

Formally, a counter machine is a tuple $M = \langle \mathbf{x}, \mathcal{L}, \ell_{init}, \ell_{fin}, \Rightarrow, \Lambda \rangle$, where $\mathbf{x}$ is a set of first-order variables ranging over $\mathbb{Z}$, $\mathcal{L}$ is a set of *control locations*, $\ell_{init}, \ell_{fin} \in \mathcal{L}$ are *initial* and *final* control locations, $\Rightarrow$ is a set of *transition rules* of the form $\ell \overset{R}{\Rightarrow} \ell'$, where $\ell, \ell' \in \mathcal{L}$ are control locations, and $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ is a relation, and $\Lambda(\ell \overset{R}{\Rightarrow} \ell')$ gives the class of R. A *loop* is a path in the control graph $\langle \mathcal{L}, \Rightarrow \rangle$ of M, where the source and the destination locations are the same, and every transition rule appears only once. A counter machine is said to be *flat* if and only if every control location is the source/destination of at most one loop. The *binary size* of a counter machine M is $\|M\|_2 = \sum_{\ell \overset{R}{\Rightarrow} \ell'} \|R\|_2^{\Lambda(\ell \overset{R}{\Rightarrow} \ell')}$.

A *configuration* of M is a pair (ℓ, ν), where $\ell \in \mathcal{L}$ is a control location, and $\nu \in \mathbb{Z}^{\mathbf{x}}$ is a valuation of the counters. A *run of M to ℓ* is a sequence of configurations $(\ell_0, \nu_0), \ldots, (\ell_k, \nu_k)$, of length $k \geq 0$, where $\ell_0 = \ell_{init}$, $\ell_k = \ell$, and for each $i = 0, \ldots, k-1$, there exists a transition rule $\ell_i \overset{R_i}{\Rightarrow} \ell_{i+1}$ such that $(\nu_i, \nu_{i+1}) \in R_i$. If ℓ is not specified, we assume $\ell = \ell_{fin}$, and say that the sequence is a *run of M*.

The *reachability problem* asks, given a counter machine M, whether *there exists a run in M?* This problem is, in general, undecidable [18], and it is decidable for flat counter machines whose loops are labeled only with certain, restricted, classes of QFPA relations, such as difference bounds (Def. 7) or octagons (Def. 9). The crux of the decidability proofs in these cases is that the transitive closure of any relation of the above type can be defined in QFPA, and is, moreover, effectively computable (see [6] for an algorithm). The goal of this paper is to provide tight bounds on the complexity of the reachability problem in these decidable cases. The parameter of the decision problem is the binary size of the input counter machine M, i.e. $\|M\|_2$. The following theorem proves decidability of the reachability problem for flat counter machines, under the assumption that the composition L of the relations on every loop in a counter machine has a *QFPA-definable transitive closure*.

Theorem 1 ([8,6,2]). *The reachability problem is decidable for any class of counter machines $\mathcal{M} = \{M$ flat counter machine $\mid$ for all $q \overset{R_1}{\Rightarrow} \ldots \overset{R_n}{\Rightarrow} q$ in M, $(R_1 \circ \ldots \circ R_n)^+$ is QFPA-definable$\}$.*

4 Periodic Relations

We introduce a notion of periodicity on classes of relations that can be naturally represented as matrices. In general, an infinite sequence of integers is said to be *periodic* if the elements of the sequence beyond a certain threshold (prefix), and which are situated at equal distance (period) one from another, differ by the same quantity (rate). This notion of periodicity is lifted to matrices of integers, entry-wise. If R is a periodic relation, the sequence of powers $\{R^k\}_{k \geq 0}$ has an infinite subsequence, that can be captured by a QFPA formula, defining infinitely many powers of the relation.

Example 1. For instance, consider the relation $R \Leftrightarrow x' = y + 1 \wedge y' = x$. This relation is periodic, and we have $R^{2k+1} \Leftrightarrow x' = y + k + 1 \wedge y' = x + k$ and $R^{2k+2} \Leftrightarrow x' = x + k + 1 \wedge y' = y + k + 1$, for all $k \geq 0$.

Definition 2. *An infinite sequence of matrices $\{A_k \in \mathbb{Z}_\infty^{m\times m}\}_{k=0}^{\infty}$ is said to be* periodic *if and only if there exist integers $b, c > 0$ and matrices $\Lambda_0, \ldots, \Lambda_{c-1} \in \mathbb{Z}_\infty^{m\times m}$ such that $A_{b+(k+1)c+i} = \Lambda_i + A_{b+kc+i}$, for all $k \geq 0$ and $i \in [c]$.*

The smallest integers b, c are called the *prefix* and the *period* of the sequence. The matrices Λ_i, corresponding to the prefix-period pair (b, c), are called the *rates* of the sequence. A relation R is said to be $*$*-consistent* if and only if $R^n \neq \emptyset$, for all $n > 0$.

Definition 3. *A class of relations $\mathcal{R}$ is said to be* periodic *iff there exist two functions $\sigma : \mathcal{R} \to \bigcup_{m>0} \mathbb{Z}_\infty^{m\times m}$ and $\rho : \bigcup_{m>0} \mathbb{Z}_\infty^{m\times m} \to \mathcal{R}$ such that $\rho(\sigma(\phi)) \Leftrightarrow \phi$, for each formula $\phi \in \mathcal{R}$ and for any $*$-consistent relation R defined by a formula from $\mathcal{R}$ the sequence of matrices $\{\sigma(R^i)\}_{i\geq 0}$ is periodic.*

If R is a $*$-consistent relation, the prefix, period $b, c > 0$ and rates $\Lambda_0, \ldots, \Lambda_{c-1} \in \mathbb{Z}^{m\times m}$ of the $\{\sigma(R^i)\}_{i\geq 0}$ sequence are called the *prefix*, *period* and *rates* of R, respectively. Otherwise, if R is not $*$-consistent, we convene that its prefix is the smallest $b > 0$ such that $R^b = \emptyset$, and its period is one. Examples of mappings σ and ρ are given in Section 7.3 for difference bounds relations, and in Section 8.1 for octagonal relations.

Definition 4. *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a relation. The* closed form *of R is the formula $\widehat{R}(k, \mathbf{x}, \mathbf{x}')$, where $k \notin \mathbf{x}$, such that the formula $\widehat{R}[n/k]$ defines R^n, for all $n \geq 0$.*

If $\mathcal{R}$ is a class of relations, let $\mathcal{R}[k]$ denote the set of closed forms of relations defined by formulae in $\mathcal{R}$[4]. Let $\mathbb{Z}[k]_\infty^{m\times m}$ be the set of matrices $M[k]$ of univariate linear terms, i.e. $M_{ij} \equiv a_{ij} \cdot k + b_{ij}$, where $a_{ij}, b_{ij} \in \mathbb{Z}$, for all $1 \leq i, j \leq m$ or $M_{ij} = \infty$. In addition to the σ and ρ functions from Def. 3, we consider a function $\pi : \bigcup_{m>0} \mathbb{Z}[k]_\infty^{m\times m} \to \mathcal{R}[k]$, mapping matrices $M[k]$ into formulae $\phi(k, \mathbf{x}, \mathbf{x}')$ such that $\pi(M)[n/k] \Leftrightarrow \rho(M[n/k])$, for all $n \geq 0$. The following lemma characterizes the closed form of a periodic relation, by defining an infinite periodic subsequence of powers of the form $\{R^{kc+b+i}\}_{k\geq 0}$, for some $b, c > 0$ and $i \in [c]$.

Lemma 1. *Let $\mathcal{R}$ be a periodic class of relations, and $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a $\mathcal{R}$-definable relation. Let $b, c > 0$ be integers, and Λ_i be matrices, for all $i \in [c]$. Consider the following statements, for all $k \geq 0$ and $i \in [c]$:*

1. *R is $*$-consistent*
2. $\widehat{R}(k \cdot c + b + i) \Leftrightarrow \pi(k \cdot \Lambda_i + \sigma(R^{b+i}))$
3. $\pi(k \cdot \Lambda_i + \sigma(R^{b+i})) \not\Leftrightarrow$ **false**
4. $\exists \mathbf{y} \,.\, \pi(k \cdot \Lambda_i + \sigma(R^{b+i}))(\mathbf{x}, \mathbf{y}) \wedge R^c(\mathbf{y}, \mathbf{x}') \Leftrightarrow \pi((k+1) \cdot \Lambda_i + \sigma(R^{b+i}))(\mathbf{x}, \mathbf{x}')$

Then (1) and (2) hold if and only if (3) and (4) hold.

[4] The closed form of a QFPA-definable relation can always be defined in first-order arithmetic, using Gödel's encoding of integer sequences, and is not, in general, equivalent to a QFPA formula.

5 Flat Counter Machines with Periodic Loops

For simplicity's sake, consider first the counter machines with the structure below:

$$\ell_{init} \xrightarrow{I(\mathbf{x}')} \overset{R(\mathbf{x},\mathbf{x}')}{\overset{\curvearrowright}{\ell}} \xrightarrow{F(\mathbf{x})} \ell_{fin} \tag{1}$$

where $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ is a periodic relation (Def. 3), and $I, F \subseteq \mathbb{Z}^{\mathbf{x}}$ are QFPA-definable sets of valuations. In the following, we give sufficient conditions (Def. 6) under which the reachability problem for the counter machines (1) is NP-complete.

Definition 5. *A class of relations $\mathcal{R}$ is said to be* poly-logarithmic *if and only if there exist integer constants $p,q,r,s > 0$, depending on $\mathcal{R}$, such that, for all $P,Q,R \in \mathcal{R}$:*

1. $\|R^n\|_2 = O(\|R\|_2^p \cdot (\log_2 n)^q)$*, for all $n > 0$*
2. *the composition $P \circ Q$ can be computed in time $O((\|P\|_2 + \|Q\|_2)^r)$*
3. *the consistency $R \not\Leftrightarrow$* **false** *can be checked in time $O(\|R\|_2^s)$*

If $\mathcal{R}$ is a poly-logarithmic class of relations, it is not difficult to see that there exists a constant $d > 0$, depending of $\mathcal{R}$, such that, for any $\mathcal{R}$-definable relation R, the n-th power R^n can be computed by a fast exponentiation algorithm in time $O((\|R\|_2 \cdot \log_2 n)^d)$.

Definition 6. *A class of periodic relations $\mathcal{R}$ is said to be* exponential *if and only if (A) $\mathcal{R}$ is poly-logarithmic, (B) the mappings σ, ρ and π (Def. 3) are computable in PTIME, and (C) for each $\mathcal{R}$-definable relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$:*

1. *there exist integer constants $p,q > 0$, depending on $\mathcal{R}$, such that the prefix and period of R are bounded by $2^{\|R\|_2^p}$ and $2^{\|R\|_2^q}$, respectively*
2. *given $i \in [c]$ and $\Lambda_i = \sigma(R^{b+c+i}) - \sigma(R^{b+i})$, points (3) and (4) of Lemma 1 can be checked in NPTIME($\|R\|_2$)*

The idea of the reduction is to show the existence of a non-deterministic Turing machine (Alg. 1) that produces, in time at most polynomial in the binary size of the input, a QFPA formula, which encodes the reachability question for the given counter machine. If the formula produced by a non-deterministic branch is satisfiable, the reachability question has a positive answer. Otherwise, if no branch of Alg. 1 returns "yes", the reachability question has a negative answer.

Since the formulae produced by Alg. 1 (lines 6 and 13) are of size at most polynomial in the size of the input (1), and that deciding whether a QFPA formula is satisfiable is an NP problem, it turns out that the reachability problem for the counter machines (1) is in NP. The general result is given in Thm. 2, which applies the idea used for single loop counter machines (1) to flat counter machines, in general.

To understand Alg. 1, observe first that the reachability problem for (1) can be stated as the satisfiability of the following formula: $I(\mathbf{x}) \wedge k \geq 0 \wedge \widehat{R}(k,\mathbf{x},\mathbf{x}') \wedge F(\mathbf{x}')$. Since, in general, the closed form $\widehat{R}(k,\mathbf{x},\mathbf{x}')$ is not QFPA-definable, we focus on the case where R is a periodic relation (Def. 3). We distinguish two cases. First, if R is not $*$-consistent, i.e. $R^i = \emptyset$ if and only if i is greater or equal than the prefix b of R, the reachability

Algorithm 1. Non-deterministic Algorithm for the Reachability Problem (1)

```
1: function IsReachable($I, R, F$)
2:     goto 8 or 3 [guess whether $R$ is $*$-consistent]
3:     choose $0 < b < 2^{\|R\|_2^p}$
4:     assume $R^{b-1} \neq \emptyset$ and $R^b = \emptyset$ [check that $R$ is not $*$-consistent]
5:     choose $i \in [b]$
6:     assume $\exists \mathbf{x} \exists \mathbf{x}' . I(\mathbf{x}) \wedge R^i(\mathbf{x}, \mathbf{x}') \wedge F(\mathbf{x}')$
7:     return Yes
8:     choose $0 < b < 2^{\|R\|_2^p}$, $0 < c < 2^{\|R\|_2^q}$ and $j \in [c]$
9:     $\Lambda \leftarrow \sigma(R^{b+c+j}) - \sigma(R^{b+j})$
10:    assume $\forall k \geq 0 \; \exists \mathbf{x} \exists \mathbf{x}' . \pi(k \cdot \Lambda + \sigma(R^{b+j}))(\mathbf{x}, \mathbf{x}')$ [check that $R$ is $*$-consistent]
11:    assume $\left( \begin{array}{c} \forall \mathbf{x} \forall \mathbf{x}' \forall k \geq 0 \; [\exists \mathbf{y} . \pi(k \cdot \Lambda_i + \sigma(R^{b+i}))(\mathbf{x}, \mathbf{y}) \; \wedge \; R^c(\mathbf{y}, \mathbf{x}')] \\ \Leftrightarrow \pi((k+1) \cdot \Lambda_i + \sigma(R^{b+i}))(\mathbf{x}, \mathbf{x}') \end{array} \right)$
12:    choose $i \in [b]$
13:    assume $\exists \mathbf{x} \exists \mathbf{x}' . I(\mathbf{x}) \wedge [R^i(\mathbf{x}, \mathbf{x}') \vee (k \geq 0 \wedge \pi(k \cdot \Lambda + \sigma(R^{b+j})))] \wedge F(\mathbf{x}')$
14:    return Yes
```

problem for (1) is equivalent to the satisfiability of the formula $I(\mathbf{x}) \wedge \left[\bigvee_{i=0}^{b-1} R^i(\mathbf{x}, \mathbf{x}') \right] \wedge F(\mathbf{x}')$. Second, if R is $*$-consistent, the reachability problem for (1) is equivalent to the satisfiability of the following formula:

$$I(\mathbf{x}) \wedge \Big[\underbrace{\bigvee_{i=0}^{b-1} R^i(\mathbf{x}, \mathbf{x}')}_{\text{prefix}} \vee \underbrace{\bigvee_{j=0}^{c-1} k \geq 0 \wedge \pi(k \cdot \Lambda_j + \sigma(R^{b+j}))}_{\text{period}} \Big] \wedge F(\mathbf{x}') \tag{2}$$

where $b, c > 0$ are integers, and $\Lambda_0, \ldots, \Lambda_{c-1}$ are matrices meeting the conditions of the second point of Lemma 1. The first disjunct above takes care of the case when the number of iterations of the loop is smaller than the prefix b, and the second one deals with the other case, when $kc + b + j$ iterations of the loop are needed, for some $k \geq 0$ and $j \in [c]$.

The first guess of Alg. 1 is whether R is $*$-consistent or not (line 2). If the guess was that R is not $*$-consistent, Alg. 1 guesses further a positive constant b, bounded by $2^{\|R\|_2^p}$, where $p > 0$ depends on the class $\mathcal{R}$ (line 3). Then it checks that b is the prefix of R, by computing R^{b-1} and R^b, and checking that $R^{b-1} \neq \emptyset$ and $R^b = \emptyset$ (line 4). By Def. 5, this check can be done in PTIME($\|R\|_2$). If the prefix check (line 4) is successful, the reachability problem can be encoded in QFPA by further guessing $i \in [B]$, and producing the QFPA formula $I(\mathbf{x}) \wedge R^i(\mathbf{x}, \mathbf{x}') \wedge F(\mathbf{x}')$ (line 6). Since $\mathcal{R}$ is a poly-logarithmic class, $\|R^i\|_2 = O(\|R\|_2^r \cdot (\log_2 i)^s) = O(\|R\|_2^{r+s})$, for some $r, s > 0$, depending on $\mathcal{R}$. Thus, the binary size of this formula is polynomial in $\|I\|_2 + \|R\|_2 + \|F\|_2$, and the reachability problem, can be answered in NPTIME($\|R\|_2 + \|I\|_2 + \|F\|_2$), by checking satisfiability of this formula (line 6).

If, on the other hand, the first guess was that R is $*$-consistent, then Alg. 1 will further guess constants $0 < b < 2^{\|R\|_2^p}$ and $0 < c < 2^{\|R\|_2^q}$, for some constants $p, q > 0$ depending on $\mathcal{R}$, and $j \in [c]$ (line 8). Next, it computes the powers R^{b+j} and R^{b+c+j} in

PTIME($\|R\|_2$), using fast exponentiation, and lets $\Lambda = \sigma(R^{b+c+j}) - \sigma(R^{b+j})$. Clearly, the binary size of Λ is bounded by a polynomial in $\|R\|_2$. Further, the algorithm needs to check whether the choices of b, c, j and Λ where adequate for defining the closed form of the infinite sequence of powers $\{R^{c \cdot k+b+j}\}_{k \geq 0}$, using Lemma 1. Moreover, it also needs to check the initial guess that R is $*$-consistent, using this closed form. To this end, it must check the points (3) and (4) of Lemma 1, which by Def. 6 (point C.2) can be done in NPTIME($\|R\|_2$) (lines 10 and 11 of Alg. 1, respectively). Next, Alg. 1 outputs a QFPA formula encoding the reachability problem, using the closed form for the sequence $\{R^{c \cdot k+b+j}\}_{k \geq 0}$ (line 13). The size of this formula is polynomial in $\|I\|_2 + \|R\|_2 + \|F\|_2$, and its satisfiability status, and thus the reachability problem for the counter machine (1), can be decided in NPTIME($\|I\|_2 + \|R\|_2 + \|F\|_2$).

It is not difficult to see that the reachability problem for (1) is NP-hard, by reduction from the satisfiability problem for QFPA [22]: let $I(\mathbf{x})$ be any QFPA formula over $\mathbf{x}$, $R =$ **false** and $F =$ **true**. Then q_f is reachable from q_i if and only if $I(\mathbf{x})$ is satisfiable. The following theorem generalizes the proof from (1) to general flat counter machines.

Theorem 2. *If $\mathcal{R}$ is a periodic exponential class of relations, the reachability problem for the class $\mathcal{M}_{\mathcal{R}} = \{M$ flat counter machine $\mid$ for all rules $q \overset{R}{\Rightarrow} q'$ on a loop of M, R is $\mathcal{R}$-definable$\}$ is NP-complete.*

6 The Periodicity of Tropical Matrix Powers

Weighted graphs are central to the upcoming developments. The main intuition is that the sequence of matrices representing the powers of a difference bounds relation captures *minimal weight paths* of lengths $1, 2, 3 \ldots$ in a weighted graph. Formally, a *weighted digraph* is a tuple $G = \langle V, E, w \rangle$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $w : E \to \mathbb{Z}$ is a weight function. A path π in G is said to be *elementary* if all vertices on π are distinct, except for the first and last vertex, which may be the same. For a path π, we denote its length by $|\pi|$, and its weight (the sum of the weights of all edges on π) by $w(\pi)$. The *average weight* of π is defined as $\overline{w}(\pi) = \frac{w(\pi)}{|\pi|}$. We assume that the reader is familiar with the notion of strongly connected component (SCC). A cycle is said to be *critical* if it has minimal average weight among all cycles in its SCC. The *cyclicity* of a SCC is the greatest common divisor of the lengths of all its elementary critical cycles, or 1, if the SCC contains no cycles.

Let $A \in \mathbb{Z}_\infty^{m \times m}$ be a square matrix, and G be any weighted graph, such that A is the incidence matrix of G. Let $(A \boxtimes B)_{ij} = \min_{k=1}^{m}(a_{ik} + b_{kj})$ denote the tropical product of A and B, $A^{\boxtimes^1} = A$ and $A^{\boxtimes^{k+1}} = A^{\boxtimes^k} \boxtimes A$, for all $k > 0$. The sequence $\{A^{\boxtimes^k}\}_{k=1}^{\infty}$ of tropical powers of A gives the minimal weights of the paths of lengths $k = 1, 2, \ldots$ between any two vertices in G. The following theorem shows that any sequence of tropical matrix powers is periodic, and provides an accurate characterization of its period.

Theorem 3 ([21]). *Let $A \in \mathbb{Z}_\infty^{m \times m}$ be a matrix, $G = \langle V, E, w \rangle$ be a weighted graph whose incidence matrix is A, and $W_1, \ldots, W_n$ be the partition of G in strongly connected components. The sequence $\{A^{\boxtimes^k}\}_{k=1}^{\infty}$ is periodic, and its period is lcm$(c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are the cyclicities of $W_1, \ldots, W_n$, respectively.*

The above theorem does not give an estimate on the prefix of the sequence, which is carried out by the following theorem:

Theorem 4. *Given a matrix $A \in \mathbb{Z}_\infty^{m\times m}$, the sequence $\{A^{\boxtimes^k}\}_{k=1}^{\infty}$ is periodic with prefix at most* $\max(m^4, 4 \cdot M \cdot m^6)$, *where* $M = \max\{abs(A_{ij}) \mid 1 \le i,j \le m, A_{ij} < \infty\}$.

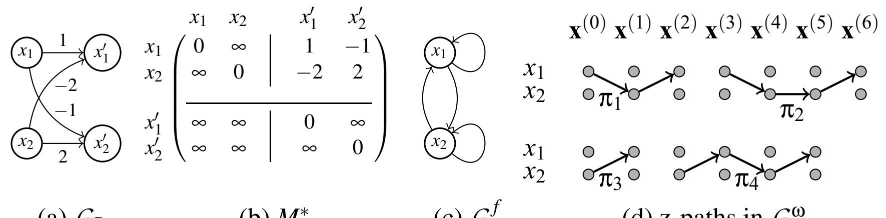
Notice that if A has only 0 and ∞ entries, then $M = 0$ and the prefix depends only on m.

7 Difference Bounds Relations

In the rest of this section, let $\mathbf{x} = \{x_1, x_2, ..., x_N\}$ be a set of variables ranging over $\mathbb{Z}$.

Definition 7. *A formula $\phi(\mathbf{x})$ is a* difference bounds constraint *if it is a finite conjunction of atomic propositions of the form $x_i - x_j \le \alpha_{ij}$, $1 \le i,j \le N$, where $\alpha_{ij} \in \mathbb{Z}$. A relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ is a* difference bounds relation *if it can be defined by a difference bounds constraint $\phi_R(\mathbf{x}, \mathbf{x}')$. The class of difference bounds relations is denoted by $\mathcal{R}_{DB}$.*

Difference bounds constraints are represented either as matrices or as graphs. If $\phi(\mathbf{x})$ is a difference bounds constraint, then a *difference bounds matrix* (DBM) representing ϕ is an $N \times N$ matrix M_ϕ such that $(M_\phi)_{ij} = \alpha_{ij}$ if $x_i - x_j \le \alpha_{ij} \in Atom(\phi)$, and $(M_\phi)_{ij} = \infty$, otherwise. The *constraint graph* $\mathcal{G}_\phi = \langle \mathbf{x}, \rightarrow \rangle$ is a weighted graph, where each vertex corresponds to a variable, and there is an edge $x_i \xrightarrow{\alpha_{ij}} x_j$ in $\mathcal{G}_\phi$ if and only if there exists a constraint $x_i - x_j \le \alpha_{ij}$ in ϕ (Fig. 1(a)). Clearly, M_ϕ is the incidence matrix of $\mathcal{G}_\phi$. If R is a difference bounds relation defined by the difference bounds constraint $\phi_R(\mathbf{x}, \mathbf{x}')$, the *folded graph* of R is the graph $\mathcal{G}_R^f = \langle \mathbf{x}, \xrightarrow{f} \rangle$, which has an edge $x_i \xrightarrow{f} x_j$ whenever $x_i \xrightarrow{\alpha} x_j$, $x_i \xrightarrow{\alpha} x'_j$, $x'_i \xrightarrow{\alpha} x_j$, or $x'_i \xrightarrow{\alpha} x'_j$ in $\mathcal{G}_R$. For any two variables $x_i, x_j \in \mathbf{x}$, we write $x_i \sim_R x_j$ whenever x_i and x_j belong to the same SCC of $\mathcal{G}_R^f$ (Fig. 1(c)). If $M \in \mathbb{Z}_\infty^{N\times N}$ is


x_1
x'_1
x_2
x'_2
1
-2
-1
2
x_1
x_2
$\mathbf{x}^{(0)}$ $\mathbf{x}^{(1)}$ $\mathbf{x}^{(2)}$ $\mathbf{x}^{(3)}$ $\mathbf{x}^{(4)}$ $\mathbf{x}^{(5)}$ $\mathbf{x}^{(6)}$
π_1
π_2
π_3
π_4

$$\begin{array}{c|cc|cc} & x_1 & x_2 & x'_1 & x'_2 \\ \hline x_1 & 0 & \infty & 1 & -1 \\ x_2 & \infty & 0 & -2 & 2 \\ \hline x'_1 & \infty & \infty & 0 & \infty \\ x'_2 & \infty & \infty & \infty & 0 \end{array}$$

(a) $\mathcal{G}_R$ (b) M_R^* (c) $\mathcal{G}_R^f$ (d) z-paths in $\mathcal{G}_R^\omega$

Fig. 1. Let $R(x_1, x_2, x'_1, x'_2) \Leftrightarrow x_1 - x'_1 \le 1 \wedge x_1 - x'_2 \le -1 \wedge x_2 - x'_1 \le -2 \wedge x_2 - x'_2 \le 2$ be a difference bounds relation. **(a)** shows the graph representation $\mathcal{G}_R$, **(b)** the closed DBM representation of R, and **(c)** the folded graph of $\mathcal{G}_R$, where $x_1 \sim_R x_2$. **(d)** shows several odd forward z-paths: π_1 (essential and repeating), π_2 (repeating), π_3 (essential) and $\pi_4 = \pi_3.\pi_1$ (neither essential nor repeating).

a DBM, we define[5]:

$$\begin{array}{ll} \Phi_M^{uu} \equiv \bigwedge_{M_{ij}<\infty} x_i - x_j \leq M_{ij} & \Phi_M^{pu} \equiv \bigwedge_{M_{ij}<\infty} x'_i - x_j \leq M_{ij} \\ \Phi_M^{up} \equiv \bigwedge_{M_{ij}<\infty} x_i - x'_j \leq M_{ij} & \Phi_M^{pp} \equiv \bigwedge_{M_{ij}<\infty} x'_i - x'_j \leq M_{ij} \end{array}$$

A DBM M is said to be *consistent* if and only if Φ_M^{uu} is consistent. A consistent difference bounds matrix $M \in \mathbb{Z}_\infty^{N\times N}$ is said to be *closed* if $M_{ii} = 0$, for all $1 \leq i \leq N$, and all triangle inequalities $M_{ik} \leq M_{ij} + M_{jk}$ hold, for all $1 \leq i,j,k \leq N$. Given a consistent DBM M, the (unique) closed DBM which is logically equivalent to M is denoted by M^* (Fig. 1(b)). It is well known that difference bounds constraints have quantifier elimination[6], and are thus closed under relational composition.

Lemma 2. *The class $\mathcal{R}_{DB}$ is poly-logarithmic.*

7.1 Zigzag Automata

Zigzag automata have been used in the proof of Presburger definability of transitive closures [8], and of periodicity [6], for difference bounds and octagonal relations. They are needed here for showing that difference bounds relations are exponential (Def. 6). Let $\mathbf{x} = \{x_1,\ldots,x_N\}$ be a set of variables, and $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a difference bounds relation, with constraint graph $\mathcal{G}_R$. Let $\Sigma_R = 2^{\mathcal{G}_R}$ denote the set of subgraphs of $\mathcal{G}_R$. A *finite word* of length $n \geq 0$ over Σ_R is a mapping $w : [n] \to \Sigma_R$. The notion of finite words over Σ_R extends naturally to infinite words $w : \mathbb{N} \to \Sigma_R$, and to bi-infinite words $w : \mathbb{Z} \to \Sigma_R$. The concatenation of two finite words $w : [n] \to \Sigma_R$ and $w' : [m] \to \Sigma_R$ is a word $w \cdot w' : [n+m] \to \Sigma_R$, defined as $(w\cdot w')(i) = w(i)$, for all $0 \leq i < n$ and $(w\cdot w')(i) = w'(i-n)$, for all $n \leq i < n+m$. The set of finite words is denoted Σ_R^*. For a finite word $w : [n] \to \Sigma_R$, we denote by ${}^\omega w^\omega$ its bi-infinite iteration, i.e. ${}^\omega w^\omega(i) = w(i \mod n)$ for all $i \in \mathbb{Z}$. For example, Fig. 2(a) shows the constraint graph $\mathcal{G}_R$ of a difference bounds relation R, and Fig. 2(b) shows several symbols $\gamma_1,\ldots,\gamma_9 \in \Sigma_R$. We associate with every finite word $w : [n] \to \Sigma$ a graph $\mathcal{H}_w = (\bigcup_{i=0}^n \mathbf{x}^{(i)}, \to)$, where $\mathbf{x}^{(i)} = \{x^{(i)} \mid x \in \mathbf{x}\}$, and:

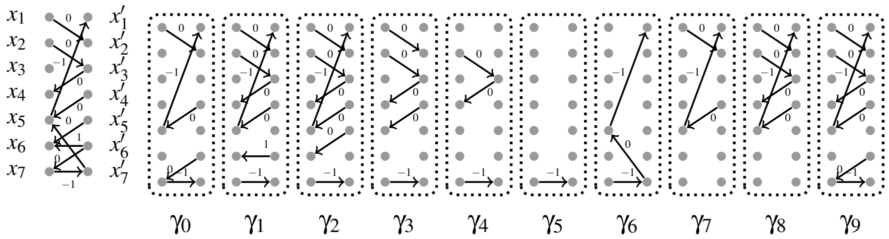
- $x_k^{(i)} \xrightarrow{\alpha} x_\ell^{(i+1)}$ in $\mathcal{H}_w$ if and only if $x_k \xrightarrow{\alpha} x'_\ell$ in $w(i)$
- $x_k^{(i+1)} \xrightarrow{\alpha} x_\ell^{(i)}$ in $\mathcal{H}_w$ if and only if $x'_k \xrightarrow{\alpha} x_\ell$ in $w(i)$

for all $1 \leq k,\ell \leq N$ and for all $0 \leq i < n$. For example, Fig. 2(c) shows the graph $\mathcal{H}_v$ corresponding to the word $v = \gamma_0.\gamma_1^2.\gamma_2.\gamma_3.\gamma_4.\gamma_5^3.\gamma_6.\gamma_7.\gamma_8^3.\gamma_9.\gamma_1.\gamma_2.\gamma_3.\gamma_4$. This notation is extended to bi-infinite words, in the obvious way. In the following, we abuse notation and denote the graph $\mathcal{H}_{{}^\omega\mathcal{G}_R{}^\omega}$, corresponding to the bi-infinite iteration of $\mathcal{G}_R$, by ${}^\omega\mathcal{G}_R{}^\omega$.

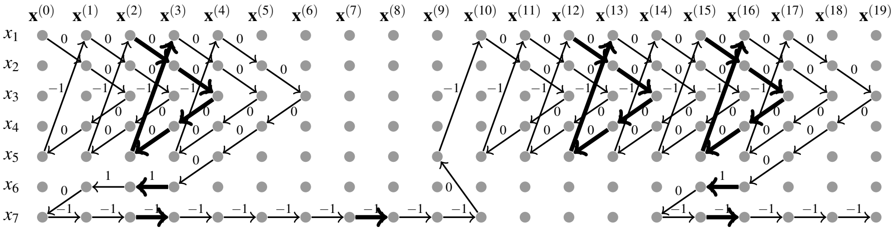
A word $w : [n] \to \Sigma_R$ is said to be *valid* if and only if each vertex of $\mathcal{H}_w$ has in-degree and out-degree at most one, and the in-degree and out-degree of each vertex from the set $\{x_k^{(i)} \mid i = 1,\ldots,n-1\}$ are equal. It is easy to see that the word v from Fig. 2(c) is valid, by inspection of the graph $\mathcal{H}_v$. The notion of validity extends from finite to bi-infinite words, in the obvious way.

[5] The superscripts u and p stand for *unprimed* and *primed*, respectively.

[6] The quantifier elimination procedure relies on the classical Floyd-Warshall closure algorithm.


x_1 x_2 x_3 x_4 x_5 x_6 x_7
x'_1 x'_2 x'_3 x'_4 x'_5 x'_6 x'_7
γ_0 γ_1 γ_2 γ_3 γ_4 γ_5 γ_6 γ_7 γ_8 γ_9

(a) $\mathcal{G}_R$ (b) Symbols of the zigzag alphabet Σ_R.

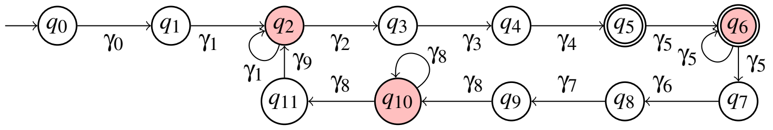

$\mathbf{x}^{(0)}$ $\mathbf{x}^{(1)}$ $\mathbf{x}^{(2)}$ $\mathbf{x}^{(3)}$ $\mathbf{x}^{(4)}$ $\mathbf{x}^{(5)}$ $\mathbf{x}^{(6)}$ $\mathbf{x}^{(7)}$ $\mathbf{x}^{(8)}$ $\mathbf{x}^{(9)}$ $\mathbf{x}^{(10)}$ $\mathbf{x}^{(11)}$ $\mathbf{x}^{(12)}$ $\mathbf{x}^{(13)}$ $\mathbf{x}^{(14)}$ $\mathbf{x}^{(15)}$ $\mathbf{x}^{(16)}$ $\mathbf{x}^{(17)}$ $\mathbf{x}^{(18)}$ $\mathbf{x}^{(19)}$
x_1 x_2 x_3 x_4 x_5 x_6 x_7

(c) A fitting odd forward z-path from $x_1^{(0)}$ to $x_7^{(19)}$ in $\mathcal{H}_{\gamma_0\cdot\gamma_1^2\cdot\gamma_2\cdot\gamma_3\cdot\gamma_4\cdot\gamma_5^3\cdot\gamma_6\cdot\gamma_7\cdot\gamma_8^2\cdot\gamma_9\cdot\gamma_1\cdot\gamma_2\cdot\gamma_3\cdot\gamma_4}$.

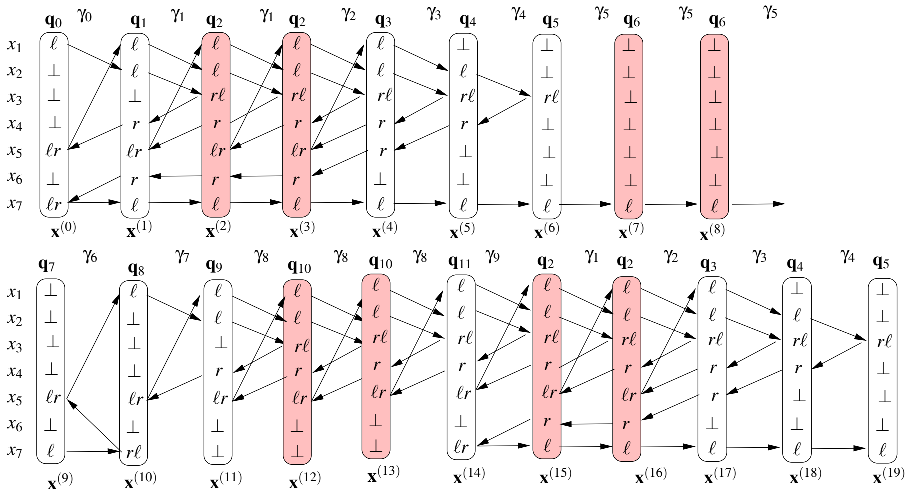
Fig. 2. Zigzag alphabet and a path in the unfolded constraint graph of a difference bounds relation $R \equiv x_1 - x'_2 \leq 0 \wedge x_2 - x'_3 \leq 0 \wedge x'_3 - x_4 \leq 0 \wedge x'_4 - x_5 \leq 0 \wedge x'_5 - x_6 \leq 0 \wedge x'_6 - x_6 \leq 1 \wedge x'_6 - x_7 \leq 0 \wedge x_7 - x'_7 \leq -1 \wedge x'_7 - x_5 \leq 0 \wedge x_5 - x'_1 \leq -1$

Given a difference bounds relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$, the set of valid finite words in Σ_R^+ is recognizable by a finite weighted automaton, called a *zigzag automaton* in the following. Let $T_R = \langle Q, \Delta, \omega \rangle$ be a weighted graph[7], called the *transition table* of the zigzag automata over Σ_R, where $Q = \{\ell, r, \ell r, r\ell, \perp\}^N$ is a set of states, $\Delta : Q \times \Sigma_R \to Q$ is a transition mapping, and $\omega : \Sigma_R \to \mathbb{Z}_\infty$ is a weight function. Intuitively, a state $\mathbf{q} = \langle \mathbf{q}_{\langle 1 \rangle}, \ldots, \mathbf{q}_{\langle N \rangle} \rangle \in Q$ describes a vertical cut in a word, as follows: for each $i = 1, \ldots, N$, $\mathbf{q}_{\langle i \rangle} = \ell$ ($\mathbf{q}_{\langle i \rangle} = r$) if there is a path in the word which traverses the cut at position i form *left* to right (*right* to left), $\mathbf{q}_{\langle i \rangle} = \ell r$ ($\mathbf{q}_{\langle i \rangle} = r\ell$) if there is a path from the right (left), which bounces to the *right* (*left*) at position i, and $\mathbf{q}_{\langle i \rangle} = \perp$ if no path in the word traverses the cut at position i (see Fig. 2(c) for an intuitive example). The transition function Δ ensures that the (local) validity condition is met. More precisely, each path $\rho : \mathbf{q}_0 \xrightarrow{\gamma_1} \mathbf{q}_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} \mathbf{q}_k$ in T_R, between two arbitrary states $\mathbf{q}_0, \mathbf{q}_k \in Q$, recognizes a valid word denoted as $\mathcal{G}_\rho = \gamma_1 \cdot \ldots \cdot \gamma_k$. The weight $\omega(\gamma)$ of a graph $\gamma \in \Sigma_R$ is the sum of the weights of its edges, and the weight of a path is $\omega(\rho) = \sum_{i=1}^k \omega(\gamma_i)$. Finally, a *zigzag automaton* is a tuple $A = \langle T_R, I, F \rangle$, where $I, F \subseteq Q$ are sets of initial and final states, respectively. We denote the *language* of A as $\mathcal{L}(A) = \{\mathcal{G}_\rho \mid q_i \overset{\rho}{\Rightarrow} q_f, q_i \in I, q_f \in F\}$. For example, the zigzag automaton depicted in Fig. 3(a), with initial state q_0 and final state q_6 has a run over the word $\gamma_0 \cdot \gamma_1^2 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \gamma_5^3 \cdot \gamma_6 \cdot \gamma_7 \cdot \gamma_8^2 \cdot \gamma_9 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4$ (see Fig. 2(c)),

[7] For reasons of presentation, we differ slightly from the definition of a weighted graph given in the previous section – here the weight of an edge is associated with the symbol labeling that edge.



(a) The zigzag automaton $A^{of}_{1,7}$ recognizing odd forward z-paths from x_1 to x_7.



(b) A run of A^{of}_{17} over $\gamma_0.\gamma_1^2.\gamma_2.\gamma_3.\gamma_4.\gamma_5^3.\gamma_6.\gamma_7.\gamma_8^3.\gamma_9.\gamma_1.\gamma_2.\gamma_3.\gamma_4$ (Fig. 2(c))

Fig. 3. Zigzag automaton for the difference bounds relation $R \equiv x_1 - x_2' \leq 0 \wedge x_2 - x_3' \leq 0 \wedge x_3' - x_4 \leq 0 \wedge x_4' - x_5 \leq 0 \wedge x_5' - x_6 \leq 0 \wedge x_6' - x_6 \leq 1 \wedge x_6' - x_7 \leq 0 \wedge x_7 - x_7' \leq -1 \wedge x_7' - x_5 \leq 0 \wedge x_5 - x_1' \leq -1$ and an example of its run (Fig. 2 contd.)

depicted in Fig. 3(b). A detailed definition of zigzag automata can be found in [8]. For the purposes of the upcoming developments, we rely on the example in Fig. 3 to give the necessary intuition.

Remark 1. The transition table $T_R = \langle Q, \Delta, \omega \rangle$ of a difference bounds relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ has at most $5^{\mathrm{card}(\mathbf{x})}$ vertices, since $Q = \{\ell, r, \ell r, r\ell, \bot\}^{\mathrm{card}(\mathbf{x})}$ is a possible representation of the set of states [8].

7.2 Paths Recognizable by Zigzag Automata

This section studies the paths that occur within the words recognizable by zigzag automata. Consider the bi-infinite unfolding of $\mathcal{G}_R$, denoted as ${}^{\omega}\mathcal{G}_R^{\omega}$. A finite path $\rho : x_{i_1}^{(j_1)} \xrightarrow{\alpha_1} x_{i_2}^{(j_2)} \xrightarrow{\alpha_2} \ldots x_{i_{k-1}}^{(j_{k-1})} \xrightarrow{\alpha_{k-1}} x_{i_k}^{(j_k)}$ in ${}^{\omega}\mathcal{G}_R^{\omega}$, for $j_1, \ldots, j_k \in \mathbb{Z}$ is said to be a *z-path* whenever, for all $1 \leq p < q \leq k$, $i_p = i_q$ and $j_p = j_q$ only if $p = 1$ and $q = k$. See Fig. 1(d) or Fig. 2(c) for examples of z-paths. We say that a variable x_{i_s} *occurs* on ρ at position j_s, for all $1 \leq s \leq k$. A z-path is called a *z-cycle* if $i_1 = i_k$ and $j_1 = j_k$. A z-path is said to be *odd* if $j_1 \neq j_k$ and *even* otherwise. For instance, in

Fig. 2(c), the z-path $x_1^{(1)} \xrightarrow{0} x_2^{(2)} \xrightarrow{0} x_3^{(3)} \xrightarrow{0} x_4^{(2)} \xrightarrow{0} x_5^{(1)} \xrightarrow{-1} x_1^{(2)}$ is an odd z-path, while $x_1^{(1)} \xrightarrow{0} x_2^{(2)} \xrightarrow{0} x_3^{(3)} \xrightarrow{0} x_4^{(2)} \xrightarrow{0} x_5^{(1)}$ is an even z-path. We denote by $\|\rho\| = \mathrm{abs}(j_k - j_1)$ its *relative* length, by $w(\rho) = \sum_{i=1}^{k-1} \alpha_i$ its *weight*, and by $\overline{\overline{w}}(\rho) = \frac{w(\rho)}{\|\rho\|}$ its *relative weight.* We write $vars(\rho)$ for the set $\{x_{i_1}, \ldots, x_{i_k}\}$ of variables occurring within ρ, called the *support set* of ρ.

An even z-path is said to be *forward* if $j_1 = j_k = \min(j_1, \ldots, j_k)$ and *backward* if $j_1 = j_k = \max(j_1, \ldots, j_k)$. An even z-path is said to be *fitting* if it is either forward or backward. An odd z-path is said to be *forward* if $j_1 < j_k$ and *backward* if $j_1 > j_k$. An odd forward (backward) z-path is said to be *fitting* if $j_1 = \min(j_1, \ldots, j_k)$ and $j_k = \max(j_1, \ldots, j_k)$ ($j_1 = \max(j_1, \ldots, j_k)$ and $j_k = \min(j_1, \ldots, j_k)$). We say that a fitting z-path ρ is *encoded* by a word w, if and only if w consists of nothing but ρ and several z-cycles not intersecting with ρ. Let $Enc(w)$ be the set of z-paths encoded by a word (this set is either a singleton or the empty set), and $Enc(\mathcal{L}) = \bigcup_{w \in \mathcal{L}} Enc(w)$ for any set of words $\mathcal{L} \subseteq \Sigma_R^*$. For instance, the word $\gamma_0.\gamma_1^2.\gamma_2.\gamma_3.\gamma_4.\gamma_5^3.\gamma_6.\gamma_7.\gamma_8^3.\gamma_9.\gamma_1.\gamma_2.\gamma_3.\gamma_4$ encodes the z-path $x_1^{(0)} \to \ldots \to x_7^{(19)}$ from in Fig. 2(c).

Theorem 5 ([8]). *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a $*$-consistent difference bounds relation, where $\mathbf{x} = \{x_1, \ldots, x_N\}$, and $\mathcal{G}_R$ be its corresponding constraint graph. Then, for every $x_i, x_j \in \mathbf{x}$, there exist zigzag automata*[8] *$A_{ij}^{\bullet} = \langle T_R, I_{ij}^{\bullet}, F_{ij}^{\bullet} \rangle$, $\bullet \in \{ef, eb, of, ob\}$, where $T_R = \langle Q, \Delta, \omega \rangle$, such that $Enc(\mathcal{L}(A_{ij}^{\bullet}))$ are the sets of fitting even/odd, forward/backward z-paths, starting with $x_i^{(k)}$ and ending with $x_j^{(\ell)}$, respectively, for some $k, \ell \in \mathbb{Z}$. Moreover, for each fitting z-path* ρ, $\omega(\rho) = \min\{\omega(\gamma) \mid \gamma \in \mathcal{L}(A_{ij}^{ef}) \cup \mathcal{L}(A_{ij}^{eb}) \cup \mathcal{L}(A_{ij}^{of}) \cup \mathcal{L}(A_{ij}^{ob}), \rho \in Enc(\gamma)\}$.

In the following, we denote the concatenation of two z-paths π and ρ by $\pi.\rho$. Notice that $\pi.\rho$ is defined only if the last variable from the first z-path equals the first variable from the second z-path, and the two z-paths do not intersect in some vertex which occurs in the middle of one of them. A z-path π is said to be *repeating* if and only if the *i*-times concatenation of π with itself, denoted π^i, is defined, for any $i > 0$. If π is repeating, then it clearly starts and ends with the same variable, and is necessarily odd. A repeating z-path is said to be *essential* if all variables occurring on the path are distinct, with the exception of the first and last, which must be equal. The concatenation of an essential repeating z-path with itself several times is called an *essential power*. For instance, in Fig. 1(d) the z-path π_1 is essential and repeating, while π_2 is repeating but not essential. For a repeating z-path π, we denote by ${}^{\omega}\pi^{\omega}$ the bi-infinite concatenation of π with itself.

7.3 The Complexity of Acceleration for Difference Bounds Relations

In this section, we prove that difference bounds constraints induce a periodic exponential class of relations (Def. 6). First, we recall that difference bounds relations are periodic (Def. 3) [6]. If $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$ is a difference bounds relation, let $\sigma(R) \equiv M_R$ and,

[8] Superscripts ef, eb, of and ob stand for *even forward, even backward, odd forward* and *odd backward*, respectively.

for each $M \in \mathbb{Z}_\infty^{2N\times 2N}$, let ${}^{\blacksquare}M$, ${}_{\blacksquare}M$, $M^{\blacksquare}$, $M_{\blacksquare} \in \mathbb{Z}^{N\times N}$ denote its top-left, bottom-left, top-right and bottom-right corners, respectively. Intuitively, ${}^{\blacksquare}M$, ${}_{\blacksquare}M$, $M^{\blacksquare}$, $M_{\blacksquare}$ capture constraints of the forms $x_i - x_j \leq c$, $x'_i - x_j \leq c$, $x_i - x'_j \leq c$ and $x'_i - x'_j \leq c$, respectively (see Fig. 1(b)). We define $\rho(M) \equiv \Phi^{uu}_{{}^{\blacksquare}M} \wedge \Phi^{up}_{M^{\blacksquare}} \wedge \Phi^{pu}_{{}_{\blacksquare}M} \wedge \Phi^{pp}_{M_{\blacksquare}}$. If $M \in \mathbb{Z}[k]_\infty^{2N\times 2N}$ is a matrix of univariate linear terms in k, $\pi(M)(k,\mathbf{x},\mathbf{x}')$ is defined analogously to ρ.

With these definitions, it was shown in [6], that the class of difference bounds relations is periodic (Def. 3). The reason is that the sequence of difference bounds matrices $\{M_{R^i}\}_{i=1}^\infty$ corresponding to the powers of a relation R is a pointwise projection of the sequence of tropical powers $\{\mathcal{M}_R^{\boxtimes^i}\}_{i=1}^\infty$ of the incidence matrix $\mathcal{M}_R$ of the transition table T_R. By Thm. 3, any sequence of tropical powers of a matrix is periodic, which entails the periodicity of the difference bounds relation R. Recall that the number of vertices in T_R is $5^N = 2^{O(N)}$. Consequently, the prefix of a difference bounds relation can be bounded using Thm. 4:

Lemma 3. *The prefix of a difference bounds relation $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$ is $2^{O(\|R\|_2)}$.*

A preliminary estimation of the upper bound of the period of a difference bounds relation $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$ can be already done using Thm. 3. Since the size of the transition table T_R of the zigzag automata for R is bounded by 5^N, by definition, the cyclicity of any SCC of T_R is at most 5^N, hence, by Thm. 3, the period is bounded by $lcm(1,\ldots,5^N)$. Applying the following lemma, one shows immediately that the period is $2^{2^{O(N)}}$.

Lemma 4. *For each $n \geq 1$, $lcm(1,\ldots,n) = 2^{O(n)}$.*

We next improve the bound on periods to simply exponential (Thm. 6).

Theorem 6. *The period of a difference bounds relation $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$ is $2^{O(N)}$.*

This leads to one of the main results of the paper:

Theorem 7. *The class $\mathcal{R}_{DB}$ is exponential, and the reachability problem for the class $\mathcal{M}_{DB} = \{M$ flat counter machine $\mid$ for all $q \xrightarrow{R} q'$ on a loop of M, R is $\mathcal{R}_{DB}$-definable$\}$ is NP-complete.*

Before proceeding with the technical developments, we summarize the proof idea of Thm 6. Let T_R be the transition table of the zigzag automata for the difference bounds relation $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$, and let $\mathcal{M}_R$ be its incidence matrix. The main idea is that each non-trivial SCC of T_R, which intersects a path between an initial and a final state of a zigzag automaton, contains a *critical elementary cycle* λ, whose length divides $lcm(1,\ldots,N)$. Then the cyclicity of the SCC containing λ is, by definition, the greatest common divisor of the lengths of all critical elementary cycles of the SCC, and consequently, a divisor of $lcm(1,\ldots,N)$ as well. Since this holds for any non-trivial SCC in T_R, by Thm. 3, the period of the sequence $\{\mathcal{M}_R^{\boxtimes^k}\}_{k=1}^\infty$ of tropical powers of $\mathcal{M}_R$ is also a divisor of $lcm(1,\ldots,N)$, which is of the order of $2^{O(N)}$ (Lemma 4).

It remains to prove the existence, in each non-trivial SCC of T_R, of an elementary critical cycle of length which divides $lcm(1,\ldots,N)$. The proof consists of several steps:

1. Let $q \xrightarrow{\gamma} q$ be a critical cycle of T_R. Intuitively, a sufficiently long iteration of γ will exhibit a word z, consisting of *repeating z-paths* (and possibly several cycles), such that $\overline{w}(z) = \overline{w}(\gamma)$.

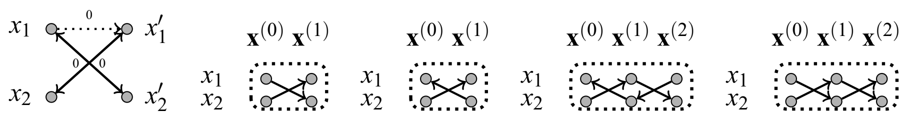
2. We define an equivalence relation on repeating z-paths (Def. 8), and define a word μ, which is obtained from z by keeping only one representative per equivalence class. Moreover, we have $\overline{w}(\mu) \leq \overline{w}(z)$ (Lemma 5), and we show that it is possible to connect μ to z both left and right, via some connecting words η and ξ, thus obtaining a valid word $z^m.\eta.\mu^n.\xi.z^p$, for every $m,n,p > 0$ (Lemma 6).
3. The word μ is further used to define a word λ, consisting only of essential powers $\pi_1^{n_1},\ldots,\pi_k^{n_k}$, where $|\pi_i| \leq N$, for all $i = 1,\ldots,k$ such that $\overline{w}(\lambda) \leq \overline{w}(\mu)$, and there exist words σ and τ, such that $\mu^q.\sigma.\lambda^r.\tau.\mu^s$ is a valid word, for all $q,r,s > 0$. Moreover, $|\lambda|$ divides $lcm(|\pi_1|,\ldots,|\pi_k|)$, and, since λ consists of essential powers $|\pi_i| \leq N$, for all $i = 1,\ldots,k$. Hence $|\lambda|$ divides $lcm(1,\ldots,N)$.
4. Finally, for sufficiently large $m,n,p,q,r > 0$, the word $z^m.\eta.\mu^n.\sigma.\lambda^p.\tau.\mu^q.\xi.z^r$ is mapped back into a path of the form: $q \rightarrow \ell \xrightarrow{\lambda} \ell \rightarrow q$, which traverses a cycle from the same SCC as the initial cycle $q \xrightarrow{\gamma} q$ (Lemma 7).

Multipaths and Reducts. A *multipath* is a (possibly empty) finite set of z-paths from ${}^{\omega}\mathcal{G}_R{}^{\omega}$, which all start and end on the same positions (see Fig. 4). Formally, a multipath $\mu = \{\pi_1,\ldots,\pi_n\}$ is a set of z-paths such that there exist integers $k < \ell$ such that, for all $i = 1,\ldots,n$, either (i) π_i is a forward (backward) odd z-path from k to ℓ (from ℓ to k), (ii) π_i is an even z-path from k to k (ℓ to ℓ), or (iii) π_i is a z-cycle whose set of positions of variable occurrences is included in the interval $[k,\ell]$, and (iv) no two z-paths in μ intersect each other. The relative length of a multipath μ, is defined as $\|\mu\| = \ell - k$ if $\mu \neq \emptyset$, or $\|\mu\| = 0$ if $\mu = \emptyset$.

For a multipath μ, we denote by μ^{ac} the set of acyclic z-paths in μ. The weight of μ is defined as $w(\mu) = \sum_{\pi\in\mu}^{n} w(\pi)$, and its average weight is $\overline{\overline{w}}(\mu) = \frac{w(\mu)}{\|\mu\|}$ if $\|\mu\| \neq 0$, or $\overline{\overline{w}}(\mu) = 0$ if $\|\mu\| = 0$. The support set of a multipath is denoted as $vars(\mu) = \bigcup_{\pi\in\mu} vars(\pi)$. The concatenation $\mu_1.\mu_2$ of two multipaths μ_1 and μ_2 is defined as the union of the two graphs, only if the result is a valid multipath. A multipath μ is *iterable* if it can be concatenated with itself any number of times, i.e. μ^i is a valid multipath, for all $i > 0$ (Fig. 4 (b,d,e)). A *repeating multipath* is an iterable multipath in which all acyclic z-paths are repeating (Fig. 4 (d,e)) – an empty multipath is repeating, by convention. A repeating multipath is said to be *essential* if every acyclic z-path is an essential power. A multipath μ is said to be *fitting* if every acyclic z-path in μ is fitting (Fig. 4 (b-e)).

Definition 8. *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a difference bounds relation, and $\mathcal{G}_R$ be its constraint graph. Let π_1 and π_2 be repeating z-paths in ${}^{\omega}\mathcal{G}_R^{\omega}$. We say that* π_1 may join π_2*, denoted $\pi_1 \bowtie_R \pi_2$, if and only if (i) there exists an SCC S of the folded graph $\mathcal{G}_R^f$, such that $vars(\pi_1) \cup vars(\pi_2) \subseteq S$ and (ii) there exists a path in ${}^{\omega}\mathcal{G}_R^{\omega}$ from some vertex in ${}^{\omega}\pi_1{}^{\omega}$ to some vertex in ${}^{\omega}\pi_2{}^{\omega}$.*

It is not hard to show that $\bowtie_R$ is an equivalence relation. For a repeating multipath μ, we denote by $\mu^{ac}_{/\bowtie_R}$ the partition of the set of acyclic paths μ^{ac} in equivalence classes of the $\bowtie_R$ relation. An *sc-multipath* (for strongly connected multipath) is a repeating multipath whose repeating z-paths belong to the same equivalence class of the $\bowtie_R$ relation (see Fig. 4). A repeating multipath ν is said to be a *reduct* of a repeating multipath μ if and



(a) $\mathcal{G}_R$ (b) μ_1 (c) μ_2 (d) μ_3 (e) μ_4

Fig. 4. Examples of multipaths. R is $x_1 = x'_2 \wedge x_2 = x'_1$ and $\mathcal{G}_R$ is shown in **(a)**. μ_1 is iterable but not repeating, μ_2 is not iterable. Both μ_3 and μ_4 are fitting, iterable, repeating, and they consist of two balanced sc-multipaths each. If R is $x_1 = x'_2 \wedge x_2 = x'_1 \wedge x_1 \leq x'_1$ instead (the dotted edge $x_1 \xrightarrow{0} x'_1$), then μ_3 is a balanced sc-multipath and μ_4 is an unbalanced sc-multipath, since $\tau_1 \bowtie_R \tau_2$ for the two forward repeating z-paths $\tau_1, \tau_2 \in \mu_4$.

only if $\nu \subseteq \mu$ and, for each equivalence class $C \in \mu^{ac}_{/\bowtie_R}$: if the difference between the number of repeating forward (backward) z-paths and the number of repeating backward (forward) z-paths in C equals $k \geq 0$, then $\nu \cap C$ contains exactly k repeating forward (backward) z-paths and no repeating backward (forward) z-path.

Example 2. Consider for instance, in Fig. 2(c), the highlighted sc-multipath $\mu = \{\pi_1 : x_1^{(2)} \xrightarrow{0} x_2^{(3)} \xrightarrow{0} x_3^{(4)} \xrightarrow{0} x_4^{(3)} \xrightarrow{0} x_5^{(2)} \xrightarrow{-1} x_1^{(3)}, \pi_2 : x_6^{(3)} \xrightarrow{1} x_6^{(2)}, \pi_3 : x_7^{(2)} \xrightarrow{-1} x_7^{(3)}\}$. Notice that $\pi_1 \bowtie_R \pi_2 \bowtie_R \pi_3$, since all variables $x_1, \ldots, x_7$ are in the same SCC of the folded graph $\mathcal{G}_R^f$ of the difference bounds relation, and, e.g. $x_5^{(5)} \xrightarrow{0} x_6^{(4)}$ connects ${}^{\omega}\pi_1{}^{\omega}$ to ${}^{\omega}\pi_2{}^{\omega}$, while $x_6^{(2)} \xrightarrow{0} x_7^{(1)}$ connects ${}^{\omega}\pi_2{}^{\omega}$ to ${}^{\omega}\pi_3{}^{\omega}$ in ${}^{\omega}\mathcal{G}_R{}^{\omega}$. Moreover, since π_1, π_3 are forward z-paths, and π_2 is a backward z-path, $\nu_1 = \{\pi_1\}$ and $\nu_2 = \{\pi_3\}$ are the only reducts of μ.

Lemma 5. *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a $*$-consistent difference bounds relation, and $\mathcal{G}_R$ be its constraint graph. Let μ be an sc-multipath in ${}^{\omega}\mathcal{G}_R^{\omega}$ and ν be a reduct of μ. Then $\overline{\overline{w}}(\nu) \leq \overline{\overline{w}}(\mu)$.*

Example 3. (contd. from Ex. 2) For instance, for the multipaths μ, ν_1 and ν_2 from Ex. 2, we have $\overline{\overline{w}}(\nu_1) = \overline{\overline{w}}(\nu_2) = \overline{\overline{w}}(\mu) = -1$. See the highlighted edges in Fig. 2(c).

Balanced SC-Multipaths and Strongly Connected Zigzag Cycles. An sc-multipath μ is said to be *balanced* if and only if the difference between the number of forward repeating and backward repeating z-paths in μ is either 1, 0, or -1. Let us observe that each reduct of a balanced sc-multipath contains at most one repeating z-path. For instance, the multipath μ from Ex. 2 is balanced, and its reducts ν_1 and ν_2 contain one z-path each.

Lemma 6. *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a $*$-consistent difference bounds relation, $\mathcal{G}_R$ be its constraint graph and μ be a balanced sc-multipath in ${}^{\omega}\mathcal{G}_R^{\omega}$. Then there exists an essential sc-multipath, $\tau = \{\tau_0\}$, such that τ_0 is an essential repeating z-path, $\overline{\overline{w}}(\tau) \leq \overline{\overline{w}}(\mu)$, and two sc-multipaths ξ and ζ such that $\mu^m.\xi.\tau^n.\zeta.\mu^p$ is a valid sc-multipath for all $m, n, p \geq 0$.*

Example 4. (contd. from Ex. 2) For instance, the multipath μ from Ex. 2 can be connected with its reducts ν_1 and ν_2, and back (see Fig. 2(c)).

The motivation for defining and studying balanced sc-multipaths can be found when examining the words generated by the iterations of a cycle $q \xrightarrow{\gamma} q$ in a zigzag automaton. Without losing generality, we assume that the state q is both *reachable* (from an initial state) and *co-reachable* (a final state is reachable from q). With this assumption, it is possible to prove that sufficiently many iterations of the γ cycle will exhibit a subgraph composed only of balanced sc-multipaths. Details can be found in [7].

Example 5. (contd. from Ex. 2) For instance, for the γ_1 cycle in the zigzag automaton in Fig. 3(a), the balanced sc-multipath is μ, defined in Ex. 2, and highlighted in Fig. 2(c), and the connecting multipaths are $\eta = \{x_2^{(3)} \xrightarrow{0} x_3^{(4)} \xrightarrow{0} x_4^{(3)}\}$ and $\xi = \{x_1^{(3)} \xrightarrow{0} x_2^{(4)}, x_4^{(4)} \xrightarrow{0} x_5^{(3)} \xrightarrow{-1} x_1^{(4)}, x_6^{(4)} \xrightarrow{1} x_6^{(3)}, x_7^{(3)} \xrightarrow{-1} x_7^{(4)}\}$. We have $\gamma_1^n = \eta.\mu^{n-2}.\xi$, for all $n \geq 2$.

The next lemma maps this graph, composed only of balanced sc-multipaths, back into another critical elementary loop $q' \xrightarrow{\lambda} q'$ of the zigzag automaton, belonging to the same SCC as γ, such that λ is composed of essential powers, and $\overline{w}(\lambda) = \overline{w}(\gamma)$. Since λ is composed of essential powers, and the length of an essential power is bounded by the number of variables N in the arithmetic representation of R, we have that $|\lambda|$ is a divisor of $lcm(1,\ldots,N)$. This is the final step needed to conclude the proof of Thm. 6.

Lemma 7. *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ be a difference bounds relation, where $\mathbf{x} = \{x_1,\ldots,x_N\}$, $T_R = \langle Q,\Delta,\omega\rangle$ be its transition table, and $A = \langle T_R,I,F\rangle$ be one of the zigzag automata from Thm. 5. If $q \in Q$ is a reachable and co-reachable state of A, and $q \xrightarrow{\gamma} q$ is a cycle, then there exists a state $q' \in Q$, a cycle $q' \xrightarrow{\lambda} q'$, and paths $q \rightarrow q'$ and $q' \rightarrow q$ in T_R, such that (i) $\overline{w}(\lambda) \leq \overline{w}(\gamma)$, and (ii) $|\lambda| \mid lcm(1,\ldots,N)$.*

Example 6. (contd. from Ex. 2 and 5) Consider the zigzag automaton depicted in Fig. 3(a). The (reachable and co-reachable) cycle $\mathbf{q}_2 \xrightarrow{\gamma_1} \mathbf{q}_2$ is a critical cycle of average weight -1. The balanced sc-multipath μ, defined in Ex. 2 is obtained by the unfolding of the $\mathbf{q}_2 \xrightarrow{\gamma_1} \mathbf{q}_2$ cycle, and has relative average weight of -1 as well. The reduct ν_1 of μ (Ex. 2) consists of one essential repeating path $\pi_1 : x_1^{(2)} \xrightarrow{0} x_2^{(3)} \xrightarrow{0} x_3^{(4)} \xrightarrow{0} x_4^{(3)} \xrightarrow{0} x_5^{(2)} \xrightarrow{-1} x_1^{(3)}$, which appears in the unfolding of another critical cycle $\mathbf{q}_{10} \xrightarrow{\gamma_8} \mathbf{q}_{10}$ of the zigzag automaton. Moreover, the latter cycle is from the same SCC as $\mathbf{q}_2 \xrightarrow{\gamma_1} \mathbf{q}_2$. The fact that both cycles belong to the same SCC is witnessed by the fact that the multipath μ can be connected to its reduct ν_1, and back, via two connecting multipaths.

8 Octagonal Relations

The class of integer octagonal constraints is defined as follows:

Definition 9. *A formula $\phi(\mathbf{x})$ is an* octagonal constraint *if it is a finite conjunction of terms of the form $x_i - x_j \leq a_{ij}$, $x_i + x_j \leq b_{ij}$ or $-x_i - x_j \leq c_{ij}$ where $a_{ij}, b_{ij}, c_{ij} \in \mathbb{Z}$, for all $1 \leq i,j \leq N$. A relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$ is an* octagonal relation *if it can be defined by an octagonal constraint $\phi_R(\mathbf{x},\mathbf{x}')$.*

We represent octagons as difference bounds constraints over the dual set of variables $\mathbf{y} = \{y_1, y_2, \ldots, y_{2N}\}$, with the convention that y_{2i-1} stands for x_i and y_{2i} for $-x_i$, respectively. For example, the octagonal constraint $x_1 + x_2 = 3$ is represented as $y_1 - y_4 \leq 3 \wedge y_2 - y_3 \leq -3$. In order to handle the $\mathbf{y}$ variables in the following, we define $\bar{\imath} = i-1$, if i is even, and $\bar{\imath} = i+1$ if i is odd. Obviously, we have $\bar{\bar{\imath}} = i$, for all $i \in \mathbb{N}$. We denote by $\overline{\phi}(\mathbf{y})$ the difference bounds constraint over $\mathbf{y}$ that represents $\phi(\mathbf{x})$:

Definition 10. *Given an octagonal constraint* $\phi(\mathbf{x})$*,* $\mathbf{x} = \{x_1, \ldots, x_N\}$*, its* difference bounds representation $\overline{\phi}(\mathbf{y})$*, over* $\mathbf{y} = \{y_1, \ldots, y_{2N}\}$*, is a conjunction of the following difference bounds constraints, where* $1 \leq i, j \leq N$*,* $c \in \mathbb{Z}$*.*

$$\begin{array}{lll}
(x_i - x_j \leq c) \in Atom(\phi) & \Leftrightarrow & (y_{2i-1} - y_{2j-1} \leq c), (y_{2j} - y_{2i} \leq c) \in Atom(\overline{\phi}) \\
(-x_i + x_j \leq c) \in Atom(\phi) & \Leftrightarrow & (y_{2j-1} - y_{2i-1} \leq c), (y_{2i} - y_{2j} \leq c) \in Atom(\overline{\phi}) \\
(-x_i - x_j \leq c) \in Atom(\phi) & \Leftrightarrow & (y_{2i} - y_{2j-1} \leq c), (y_{2j} - y_{2i-1} \leq c) \in Atom(\overline{\phi}) \\
(x_i + x_j \leq c) \in Atom(\phi) & \Leftrightarrow & (y_{2i-1} - y_{2j} \leq c), (y_{2j-1} - y_{2i} \leq c) \in Atom(\overline{\phi})
\end{array}$$

An octagonal constraint ϕ is equivalently represented by the DBM $M_{\overline{\phi}} \in \mathbb{Z}_\infty^{2N \times 2N}$, corresponding to $\overline{\phi}$. We say that a DBM $M \in \mathbb{Z}_\infty^{2N \times 2N}$ is *coherent*[9] iff $M_{ij} = M_{\bar{\jmath}\bar{\imath}}$ for all $1 \leq i, j \leq 2N$. Dually, for a coherent DBM $M \in \mathbb{Z}_\infty^{2N \times 2N}$, we define:

$$\begin{array}{ll}
\Psi_M^{uu} & \equiv \bigwedge_{1 \leq i,j \leq N} x_i - x_j \leq M_{2i-1,2j-1} \wedge x_i + x_j \leq M_{2i-1,2j} \wedge -x_i - x_j \leq M_{2i,2j-1} \\
\Psi_M^{up} & \equiv \bigwedge_{1 \leq i,j \leq N} x_i - x'_j \leq M_{2i-1,2j-1} \wedge x_i + x'_j \leq M_{2i-1,2j} \wedge -x_i - x'_j \leq M_{2i,2j-1} \\
\Psi_M^{pu} & \equiv \bigwedge_{1 \leq i,j \leq N} x'_i - x_j \leq M_{2i-1,2j-1} \wedge x'_i + x_j \leq M_{2i-1,2j} \wedge -x'_i - x_j \leq M_{2i,2j-1} \\
\Psi_M^{pp} & \equiv \bigwedge_{1 \leq i,j \leq N} x'_i - x'_j \leq M_{2i-1,2j-1} \wedge x'_i + x'_j \leq M_{2i-1,2j} \wedge -x'_i - x'_j \leq M_{2i,2j-1}
\end{array}$$

A coherent DBM M is said to be *octagonal-consistent* if and only if Ψ_M^{uu} is consistent.

Definition 11. *An octagonal-consistent coherent DBM* $M \in \mathbb{Z}_\infty^{2N \times 2N}$ *is said to be* tightly closed *iff it is closed and, for all* $1 \leq i, j \leq 2N$*,* $M_{i\bar{\imath}}$ *is even, and* $M_{ij} \leq \lfloor \frac{M_{i\bar{\imath}}}{2} \rfloor + \lfloor \frac{M_{\bar{\jmath}j}}{2} \rfloor$.

Intuitively the conditions of Def. 11 ensure that all knowledge induced by the triangle inequality and the $y_{2i-1} = -y_{2i}$ constraints has been propagated in the DBM. Given an octagonal-consistent coherent DBM $M \in \mathbb{Z}^{2N} \times \mathbb{Z}^{2N}$, we denote the (unique) logically equivalent tightly closed DBM by M^t. Octagonal constraints are closed under existential quantification, thus octagonal relations are closed under composition [4]. Tight closure of octagonal-consistent DBMs is needed for quantifier elimination. The set of octagonal constraints forms therefore a class, denoted further $\mathcal{R}_{OCT}$.

Lemma 8. *The class* $\mathcal{R}_{OCT}$ *is poly-logarithmic.*

8.1 The Complexity of Acceleration for Octagonal Relations

The proof idea for the periodicity of $\mathcal{R}_{OCT}$ is the following. Since any power R^i of an octagonal relation R is obtained by quantifier elimination, and since quantifier elimination for octagons uses the tight closure of the DBM representation, then the sequence

[9] DBM coherence is needed because $x_i - x_j \leq c$ can be represented as both $y_{2i-1} - y_{2j-1} \leq c$ and $y_{2j} - y_{2i} \leq c$.

$\{R^i\}_{i>0}$ is defined by the sequence $\{M^t_{\overline{R^i}}\}_{i>0}$ of tightly closed DBMs. In [6] we prove that this sequence of matrices is periodic, using the result from Thm. 8, below. If $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$ is an octagonal relation, let $\sigma(R) \equiv M_{\overline{R}}$ be the characteristic DBM of its difference bounds representation, and for a coherent DBM $M \in \mathbb{Z}_\infty^{4N\times 4N}$, we define $\rho(M) \equiv \Psi^{uu}_{\blacksquare_M} \wedge \Psi^{up}_{M\blacksquare} \wedge \Psi^{pu}_{\blacksquare M} \wedge \Psi^{pp}_{M_\blacksquare}$. Analogously, $\pi(M)$ is defined in the same way as ρ, for each matrix $M \in \mathbb{Z}[k]_\infty^{4N\times 4N}$ of univariate linear terms. With these definitions, periodicity of $\mathcal{R}_{OCT}$ has been shown in [6], using the periodicity of $\mathcal{R}_{DB}$ and the following theorem [4], establishing the following relation between $M^t_{\overline{R^m}}$ (the tightly closed octagonal DBM corresponding to the m-th iteration of R) and $M^*_{\overline{R}^m}$ (the closed DBM corresponding to the m-th iteration of the difference bounds relation $\overline{R}$), for all $m > 0$:

Theorem 8. *[4] Let $R \subseteq \mathbb{Z}^N \times \mathbb{Z}^N$, be a $*$-consistent octagonal relation. Then, for all $m > 0$ and $1 \leq i, j \leq 4N$:* $(M^t_{\overline{R^m}})_{ij} = \min\left\{ (M^*_{\overline{R}^m})_{ij}, \left\lfloor \frac{(M^*_{\overline{R}^m})_{i\bar{\imath}}}{2} \right\rfloor + \left\lfloor \frac{(M^*_{\overline{R}^m})_{\bar{\jmath}j}}{2} \right\rfloor \right\}$.

In the rest of this section, we show that the periodic class $\mathcal{R}_{OCT}$ is also exponential, which proves NP-completness of the reachability problem for flat counter machines with octagonal constraints labeling their loops.

Lemma 9. *Let $\{s_m\}_{m=1}^\infty$ and $\{t_m\}_{m=1}^\infty$ be two periodic sequences. Then the sequences $\{\min(s_m,t_m)\}_{m=1}^\infty$, $\{s_m + t_m\}_{m=1}^\infty$ and $\left\{\left\lfloor \frac{s_m}{2} \right\rfloor\right\}_{m=1}^\infty$ are periodic as well. Moreover, the prefixes and periods of these sequences are linear in the prefixes and periods of $\{s_m\}_{m=1}^\infty$ and $\{t_m\}_{m=1}^\infty$.*

A consequence of Thm. 8 and Lemma 9 is that the asymptotic bounds on the prefix and period and an octagonal relation match the ones of its difference bounds representation, which uses twice as many variables (Def. 10).

Lemma 10. *Let $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{x}}$, where $\mathbf{x} = \{x_1, \ldots, x_N\}$, be an octagonal relation. The prefix and period of R are $2^{O(\|R\|_2)}$ and $2^{O(N)}$, respectively.*

The previous lemma provides the bounds on the prefix and periods of octagonal relations, needed for the next theorem, which gives the second main result of the paper:

Theorem 9. *The class $\mathcal{R}_{DB}$ is exponential, and the reachability problem for the class $\mathcal{M}_{OCT} = \{M$ flat counter machine $|$ for all $q \xrightarrow{R} q'$ on a loop of M, R is $\mathcal{R}_{OCT}$-definable$\}$ is NP-complete.*

9 Conclusions and Future Work

We prove that the verification of reachability properties for flat counter machines with difference bounds and octagonal relations on loops is NP-complete. Future work includes the extension of this result to finite monoid affine relations [6], and the investigation of temporal logic properties of flat counter machines with transitions defined using these classes of relations.

References

1. Bansal, K., Koskinen, E., Wies, T., Zufferey, D.: Structural counter abstraction. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 62–77. Springer, Heidelberg (2013)

2. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. PhD, Univ. de Liège (1999)
3. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
4. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009)
5. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009)
6. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
7. Bozga, M., Iosif, R., Konečný, F.: Safety problems are NP-complete for flat integer programs with octagonal loops. Tech. Rep. arXiv 1307.5321 (2013), `http://arxiv.org/abs/1307.5321`
8. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundamenta Informaticae 91(2), 275–303 (2009)
9. Bozzelli, L., Pinchinat, S.: Verification of gap-order constraint abstractions of counter systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 88–103. Springer, Heidelberg (2012)
10. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)
11. Demri, S., Dhar, A.K., Sangnier, A.: On the complexity of verifying regular properties on flat counter systems, In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 162–173. Springer, Heidelberg (2013)
12. Demri, S., Dhar, A.K., Sangnier, A.: Taming past LTL and flat counter systems. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 179–193. Springer, Heidelberg (2012)
13. Demri, S., Jurdzinski, M., Lachish, O., Lazic, R.: The covering and boundedness problems for branching vector addition systems. J. Comput. Syst. Sci. 79(1), 23–38 (2013)
14. Gawlitza, T.M., Monniaux, D.: Invariant generation through strategy iteration in succinctly represented control flow graphs. Logical Methods in Computer Science 8(3) (2012)
15. Gurari, E.M., Ibarra, O.H.: The complexity of the equivalence problem for simple programs. J. ACM 28(3), 535–560 (1981)
16. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. J. ACM 25(1), 116–133 (1978)
17. Leroux, J.: Vector addition system reachability problem: a short self-contained proof. In: POPL, pp. 307–316 (2011)
18. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall (1967)
19. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. 6, 223–231 (1978)
20. Revesz, P.Z.: A closed-form evaluation for Datalog queries with integer (gap)-order constraints. Theor. Comput. Sci. 116(1&2), 117–149 (1993)
21. Schutter, B.D.: On the ultimate behavior of the sequence of consecutive powers of a matrix in the max-plus algebra. Linear Algebra and its Applications 307, 103–117 (2000)
22. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational Horn clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005)

Parameterized Model Checking of Token-Passing Systems

Benjamin Aminof[1], Swen Jacobs[2], Ayrat Khalimov[2], and Sasha Rubin[1,3,⋆]

[1] IST Austria
first.last@ist.ac.at
[2] TU Graz
first.last@iaik.tugraz.at
[3] TU Wien

Abstract. We revisit the parameterized model checking problem for token-passing systems and specifications in indexed $\mathsf{CTL}^*\backslash\mathsf{X}$. Emerson and Namjoshi (1995, 2003) have shown that parameterized model checking of indexed $\mathsf{CTL}^*\backslash\mathsf{X}$ in uni-directional token rings can be reduced to checking rings up to some *cutoff* size. Clarke et al. (2004) have shown a similar result for general topologies and indexed $\mathsf{LTL}\backslash\mathsf{X}$, provided processes cannot choose the directions for sending or receiving the token.

We unify and substantially extend these results by systematically exploring fragments of indexed $\mathsf{CTL}^*\backslash\mathsf{X}$ with respect to general topologies. For each fragment we establish whether a cutoff exists, and for some concrete topologies, such as rings, cliques and stars, we infer small cutoffs. Finally, we show that the problem becomes undecidable, and thus no cutoffs exist, if processes are allowed to choose the directions in which they send or from which they receive the token.

1 Introduction

As executions of programs and protocols are increasingly distributed over multiple CPU cores or even physically separated computers, correctness of concurrent systems is one of the primary challenges of formal methods today. Many concurrent systems consist of an arbitrary number of identical processes running in parallel. The parameterized model checking problem (PMCP) for concurrent systems is to decide if a given temporal logic specification holds irrespective of the number of participating processes.

The PMCP is undecidable in many cases. For example, it is undecidable already for safety specifications and finite-state processes communicating by passing a binary-valued token around a uni-directional ring [17,8]. However, decidability may be regained by restricting the communication primitives, the topologies under consideration (i.e., the underlying graph describing the communication paths

⋆ This work was supported by the Austrian Science Fund through grant P23499-N23 and through the RiSE network (S11403, S11405, S11406, S11407-N23); ERC Starting Grant (279307: Graph Games); Vienna Science and Technology Fund (WWTF) grants PROSEED, ICT12-059, and VRG11-005.

between the processes), or the specification language. In particular, previous results have shown that parameterized model checking can sometimes be reduced to model checking a finite number of instances of the system, up to some *cutoff* size.

For token-passing systems (TPSs) with uni-directional ring topologies, such cutoffs are known for specifications in the prenex fragment of indexed CTL^* without the next-time operator ($\mathsf{CTL}^*\backslash\mathsf{X}$) [10,8]. For token-passing in general topologies, cutoffs are known for the prenex fragment of indexed $\mathsf{LTL}\backslash\mathsf{X}$, provided that processes are not allowed to choose the direction to which the token is sent or from which it is received [5]. In this paper we generalize these results and elucidate what they have in common.

Previous Results. In their seminal paper [8], Emerson and Namjoshi consider systems where the token does not carry messages, and specifications are in prenex indexed temporal logic — i.e., quantifiers $\forall$ and $\exists$ over processes appear in a block at the front of the formula. They use the important concept of a *cutoff* — a number c such that the PMCP for a given class of systems and specifications can be reduced to model checking systems with up to c processes. If model checking is decidable, then existence of a cutoff implies that the PMCP is decidable. Conversely, if the PMCP is undecidable, then there can be no cutoff for such systems.

For uni-directional rings, Emerson and Namjoshi provide cutoffs for formulas with a small number k of quantified index variables, and state that their proof method allows one to obtain cutoffs for other quantifier prefixes. In brief, cutoffs exist for the branching-time specification language prenex indexed $\mathsf{CTL}^*\backslash\mathsf{X}$ and the highly regular topology of uni-directional rings.

Clarke et al. [5] consider the PMCP for token-passing systems arranged in general topologies. Their main result is that the PMCP for systems with arbitrary topologies and k-indexed $\mathsf{LTL}\backslash\mathsf{X}$ specifications (i.e., specifications with k quantifiers over processes in the prenex of the formula) can be reduced to combining the results of model-checking finitely many topologies of size at most $2k$ [5, Theorem 4]. Their proof implies that, for each k, the PMCP for linear-time specifications in k-indexed $\mathsf{LTL}\backslash\mathsf{X}$ and general topologies has a cutoff.

Questions. Comparing these results, an obvious question is: are there cutoffs for branching time temporal logics and arbitrary topologies (see Table 1)? Clarke et al. already give a first answer [5, Corollary 3]. They prove that there is no cutoff

Table 1. Direction-Unaware TPSs

	Uni-Ring Topologies	Arbitrary Topologies
indexed $\mathsf{LTL}\backslash\mathsf{X}$	–	[5]
indexed $\mathsf{CTL}^*\backslash\mathsf{X}$	[8]	this paper

Table 2. Direction-Aware TPSs

	Bi-Ring Topologies	Arbitrary Topologies
indexed $\mathsf{LTL}\backslash\mathsf{X}$	[7]	this paper
indexed $\mathsf{CTL}^*\backslash\mathsf{X}$	this paper	this paper

for token-passing systems with arbitrary topologies and specifications from 2-indexed $\mathsf{CTL}\backslash\mathsf{X}$. However, their proof makes use of formulas with unbounded nesting-depth of path quantifiers. This lead us to the first question.

Question 1. Is there a way to stratify k-indexed $\mathsf{CTL}^*\backslash\mathsf{X}$ such that for each level of the stratification there is a cutoff for systems with arbitrary topologies? In particular, does stratification by nesting-depth of path quantifiers do the trick?

Cutoffs for k-indexed temporal logic fragments immediately yield that for each k there is an algorithm (depending on k) for deciding the PMCP for k-indexed temporal logic. However, this does not imply that there is an algorithm that can compute the cutoff for a given k. In particular, it does not imply that PMCP for full prenex indexed temporal logic is decidable.

Question 2. For which topologies (rings, cliques, all?) can one conclude that the PMCP for the full prenex indexed temporal logic is decidable?

Finally, an important implicit assumption in Clarke et al. [5] is that processes are not *direction aware*, i.e., they cannot sense or choose in which direction the token is sent, or from which direction it is received. In contrast, Emerson and Kahlon [7] show that cutoffs exist for certain direction-aware systems in bi-directional rings (see Section 8). We were thus motivated to understand to what extent existing results about cutoffs can be lifted to direction-aware systems, see Table 2.

Question 3. Do cutoffs exist for direction-aware systems on arbitrary topologies and k-indexed temporal logics (such as $\mathsf{LTL}\backslash\mathsf{X}$ and $\mathsf{CTL}\backslash\mathsf{X}$)?

Our Contributions. In this paper, we answer the questions above, unifying and substantially extending the known cutoff results:

Answer to Question 1. Our main positive result (Theorem 7) states that for arbitrary parameterized topologies $\mathbf{G}$ there is a cutoff for specifications in k-indexed $\mathsf{CTL}^*_d\backslash\mathsf{X}$ — the cutoff depends on $\mathbf{G}$, the number k of the process quantifiers, and the nesting depth d of path quantifiers. In particular, indexed $\mathsf{LTL}\backslash\mathsf{X}$ is included in the case $d = 1$, and so our result generalizes the results of Clarke et al. [5].

Answer to Question 2. We prove (Theorem 14) that there exist topologies for which the PMCP is undecidable for specifications in prenex indexed $\mathsf{CTL}\backslash\mathsf{X}$ or $\mathsf{LTL}\backslash\mathsf{X}$. Note that this undecidability result does not contradict the existence of cutoffs (Theorem 7), since cutoffs may not be computable from k, d (see the note on decidability in Section 2.4). However, for certain topologies our positive result is constructive and we can compute cutoffs given k and d (Theorem 15). To illustrate, we show that rings have a cutoff of $2k$, cliques of $k+1$, and stars of $k+1$ (independent of d). In particular, PMCP is decidable for these topologies and specifications in prenex indexed $\mathsf{CTL}^*\backslash\mathsf{X}$.

Answer to Question 3. The results just mentioned assume that processes are not direction-aware. Our main negative result (Theorem 17) states that if processes can control at least one of the directions (i.e., choose in which direction to send or from which direction to receive) then the PMCP for arbitrary topologies and

k-indexed logic (even $\mathsf{LTL}\backslash\mathsf{X}$ and $\mathsf{CTL}\backslash\mathsf{X}$) is undecidable, and therefore does not have cutoffs. Moreover, if processes can control both in- and out-directions, then the PMCP is already undecidable for bi-directional rings and 1-indexed $\mathsf{LTL}\backslash\mathsf{X}$.

Technical contributions relative to previous work. Our main positive result (Theorem 7) generalizes proof techniques and ideas from previous results [8,5]. We observe that in both of these papers the main idea is to abstract a TPS by simulating the quantified processes exactly and simulating the movement of the token between these processes. The relevant information about the movement of the token is this: whether there is a direct edge, or a path (through unquantified processes) from one quantified process to another. This abstraction does not work for $\mathsf{CTL}^*_d\backslash\mathsf{X}$ and general topologies since the formula can express branching properties of the token movement. Our main observation is that the relevant information about the branching-possibilities of the token can be expressed in $\mathsf{CTL}^*_d\backslash\mathsf{X}$ over the topology itself. We develop a composition-like theorem, stating that if two topologies (with k distinguished vertices) are indistinguishable by $\mathsf{CTL}^*_d\backslash\mathsf{X}$ formulas, then the TPSs based on these topologies and an arbitrary process template P are indistinguishable by $\mathsf{CTL}^*_d\backslash\mathsf{X}$ (Theorem 9). The machinery involves a generalization of stuttering trace-equivalence [16], a notion of d-contraction that serves the same purpose as the connection topologies of Clarke et al. [5, Proposition 1], and also the main simulation idea of Emerson and Namjoshi [8, Theorem 2].

Our main negative result, undecidability of PMCP for direction-aware systems (Theorem 17), is proven by a reduction from the non-halting problem for 2-counter machines (as is typical in this area [8,11]). Due to the lack of space, full proofs are omitted, and can be found in the full version [1].

2 Definitions and Existing Results

Let $\mathbb{N}$ denote the set of positive integers. Let $[k]$ for $k \in \mathbb{N}$ denote the set $\{1, \dots, k\}$. The concatenation of strings u and w is written uw or $u \cdot w$.

Let AP denote a countably infinite set of *atomic propositions* or *atoms*. A *labeled transition system (LTS) over AP* is a tuple $(Q, Q_0, \Sigma, \delta, \lambda)$ where Q is the set of *states*, $Q_0 \subseteq Q$ are the *initial states*, Σ is the set of *transition labels* (also called *action labels*), $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, and $\lambda : Q \to 2^{\mathsf{AP}}$ is the *state-labeling* and satisfies that $\lambda(q)$ is finite (for every $q \in Q$). Transitions $(q, \sigma, q') \in \delta$ may be written $q \xrightarrow{\sigma} q'$.

A *state-action path* of an LTS $(Q, Q_0, \Sigma, \delta, \lambda)$ is a finite sequence of the form $q_0\sigma_0q_1\sigma_1 \dots q_n \in (Q\Sigma)^*Q$ or an infinite sequence $q_0\sigma_0q_1\sigma_1 \dots \in (Q\Sigma)^\omega$ such that $(q_i, \sigma_i, q_{i+1}) \in \delta$ (for all i). A *path* of an LTS is the projection $q_0q_1 \dots$ of a state-action path onto states Q. An *action-labeled path* of an LTS is the projection $\sigma_0\sigma_1 \dots$ of a state-action path onto transition labels Σ.

2.1 System Model (Direction-Unaware)

In this section we define the LTS P^G — it consists of replicated copies of a process P placed on the vertices of a graph G. Transitions in P^G are either internal (in which exactly one process moves) or synchronized (in which one process sends the token to another along an edge of G). The token starts with the process that is at the initial vertex of G.

Fix a countably infinite set of (local) atomic propositions $\mathsf{AP}_{\mathsf{pr}}$ (to be used by the states of the individual processes).

Process Template P. Let Σ_{int} denote a finite non-empty set of *internal-transition labels.* Define Σ_{pr} as the disjoint union $\Sigma_{\mathsf{int}} \cup \{\mathsf{rcv}, \mathsf{snd}\}$ where rcv and snd are new symbols.

A *process template* P is a LTS $(Q, Q_0, \Sigma_{\mathsf{pr}}, \delta, \lambda)$ over $\mathsf{AP}_{\mathsf{pr}}$ such that:

i) the state set Q is finite and can be partitioned into two non-empty sets, say $T \cup N$. States in T are said to *have the token.*
ii) The initial state set is $Q_0 = \{\iota_t, \iota_n\}$ for some $\iota_t \in T, \iota_n \in N$.
iii) Every transition $q \overset{\mathsf{snd}}{\to} q'$ satisfies that q has the token and q' does not.
iv) Every transition $q \overset{\mathsf{rcv}}{\to} q'$ satisfies that q' has the token and q does not.
v) Every transition $q \overset{\mathsf{a}}{\to} q'$ with $\mathsf{a} \in \Sigma_{\mathsf{int}}$ satisfies that q has the token if and only if q' has the token.
vi) The transition relation δ is total in the first coordinate: for every $q \in Q$ there exists $\sigma \in \Sigma_{\mathsf{pr}}, q' \in Q$ such that $(q, \sigma, q') \in \delta$ (i.e., the process P is non-terminating).
vii) Every infinite action-labeled path $a_0 a_1 \ldots$ is in the set $(\Sigma_{\mathsf{int}}^* \,\mathsf{snd}\, \Sigma_{\mathsf{int}}^* \,\mathsf{rcv})^\omega \cup (\Sigma_{\mathsf{int}}^* \,\mathsf{rcv}\, \Sigma_{\mathsf{int}}^* \,\mathsf{snd})^\omega$ (i.e., snd and rcv actions alternate continually along every infinite action-labeled path of P). [1]

The elements of Q are called *local states* and the transitions in δ are called *local transitions (of P).* A local state q such that the only transitions are of the form $q \overset{\mathsf{snd}}{\to} q'$ (for some q') is said to be *send-only.* A local state q such that the only transitions are of the form $q \overset{\mathsf{rcv}}{\to} q'$ (for some q') is said to be *receive-only.*

Topology G. A *topology* is a directed graph $G = (V, E, x)$ where $V = [k]$ for some $k \in \mathbb{N}$, vertex $x \in V$ is the *initial vertex*, $E \subseteq V \times V$, and $(v, v) \notin E$ for every $v \in V$. Vertices are called *process indices.*

We may also write $G = (V_G, E_G, x_G)$ if we need to disambiguate.

Token-Passing System P^G. Let $\mathsf{AP}_{\mathsf{sys}} := \mathsf{AP}_{\mathsf{pr}} \times \mathbb{N}$ be the *indexed atomic propositions.* For $(p, i) \in \mathsf{AP}_{\mathsf{sys}}$ we may also write p_i. Given a process template $P = (Q, Q_0, \Sigma_{\mathsf{pr}}, \delta, \lambda)$ over $\mathsf{AP}_{\mathsf{pr}}$ and a topology $G = (V, E, x)$, define the *token-passing system (TPS) P^G* as the finite LTS $(S, S_0, \Sigma_{\mathsf{int}} \cup \{\mathsf{tok}\}, \Delta, \Lambda)$ over atomic propositions $\mathsf{AP}_{\mathsf{sys}} := \mathsf{AP}_{\mathsf{pr}} \times \mathbb{N}$, where:

– The set S of *global states* is Q^V, i.e., all functions from V to Q. If $s \in Q^V$ is a global state then $s(i)$ denotes the local state of the process with index i.
– The set of *global initial states* S_0 consists of the unique global state $s_0 \in Q_0^V$ such that only $s_0(x)$ has the token (here x is the initial vertex of G).
– The labeling $\Lambda(s) \subset \mathsf{AP}_{\mathsf{sys}}$ for $s \in S$ is defined as follows: $p_i \in \Lambda(s)$ if and only if $p \in \lambda(s(i))$, for $p \in \mathsf{AP}_{\mathsf{pr}}$ and $i \in V$.

[1] This restriction was introduced by Emerson and Namjoshi in [8]. Our positive results that cutoffs exist also hold for a more liberal restriction (see Section 7).

- The *global transition relation* Δ is defined to consist of the set of all internal transitions and synchronous transitions:
  - An *internal transition* is an element (s, a, s') of $S \times \Sigma_{\mathsf{int}} \times S$ for which there exists a process index $v \in V$ such that
    i) $s(v) \xrightarrow{\mathsf{a}} s'(v)$ is a local transition of P, and
    ii) for all $w \in V \setminus \{v\}$, $s(w) = s'(w)$.
  - A *token-passing transition* is an element (s, tok, s') of $S \times \{\mathsf{tok}\} \times S$ for which there exist process indices $v, w \in V$ such that $(v, w) \in E$ and
    i) $s(v) \xrightarrow{\mathsf{snd}} s'(v)$ is a local transition of P,
    ii) $s(w) \xrightarrow{\mathsf{rcv}} s'(w)$ is a local transition of P, and
    iii) for every $u \in V \setminus \{v, w\}$, $s'(u) = s(u)$.

In words, the system P^G can be thought of the asynchronous parallel composition of P over topology G. The token starts with process x. At each time step either exactly one process makes an internal transition, or exactly two processes synchronize when one process sends the token to another along an edge of G.

2.2 System Model (Direction-Aware)

Inspired by direction-awareness in the work of Emerson and Kahlon [7], we extend the definition of TPS to include additional labels on edges, called *directions.* The idea is that processes can restrict which directions are used when they send or receive the token.

Fix finite non-empty disjoint sets $\mathsf{Dir}_{\mathsf{snd}}$ of *sending directions* and $\mathsf{Dir}_{\mathsf{rcv}}$ of *receiving directions.* A *direction-aware token-passing system* is a TPS with the following modifications.

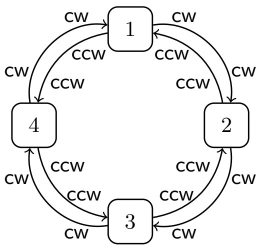
Direction-aware Topology. A *direction-aware topology* is a topology $G = (V, E, x)$ with labeling functions $\mathsf{dir}_{\mathsf{rcv}} : E \to \mathsf{Dir}_{\mathsf{rcv}}$, $\mathsf{dir}_{\mathsf{snd}} : E \to \mathsf{Dir}_{\mathsf{snd}}$.

Direction-aware Process Template. For process templates of direction-aware systems, transition labels are taken from $\Sigma_{\mathsf{pr}} := \Sigma_{\mathsf{int}} \cup \mathsf{Dir}_{\mathsf{snd}} \cup \mathsf{Dir}_{\mathsf{rcv}}$. The definition of a *direction-aware process template* is like that in Section 2.1, except that in item iii) snd is replaced by $\mathsf{d} \in \mathsf{Dir}_{\mathsf{snd}}$, in iv) rcv is replaced by $\mathsf{d} \in \mathsf{Dir}_{\mathsf{rcv}}$, and in vii) snd is replaced by $\mathsf{Dir}_{\mathsf{snd}}$ and rcv by $\mathsf{Dir}_{\mathsf{rcv}}$.

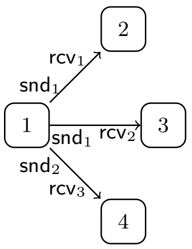
Direction-aware Token Passing System. Fix $\mathsf{Dir}_{\mathsf{snd}}$ and $\mathsf{Dir}_{\mathsf{rcv}}$, let G be a direction-aware topology and P a direction-aware process template. Define the *direction-aware token-passing system* P^G as in Section 2.1, except that token-passing transitions are now direction-aware: *direction-aware token-passing transitions* are elements (s, tok, s') of $S \times \{\mathsf{tok}\} \times S$ for which there exist process indices $v, w \in V$ with $(v, w) \in E$, $\mathsf{dir}_{\mathsf{snd}}(v, w) = \mathsf{d}$, and $\mathsf{dir}_{\mathsf{rcv}}(v, w) = \mathsf{e}$, such that:

i) $s(v) \xrightarrow{\mathsf{d}} s'(v)$ is a local transition of P.
ii) $s(w) \xrightarrow{\mathsf{e}} s'(w)$ is a local transition of P.
iii) For every $u \in V \setminus \{v, w\}$, $s'(u) = s(u)$.

Notations $\mathcal{P}_{\mathsf{u}}$, $\mathcal{P}_{\mathsf{snd}}$, $\mathcal{P}_{\mathsf{rcv}}$, $\mathcal{P}_{\mathsf{sndrcv}}$. Let $\mathcal{P}_u$ denote the set of all process templates for which $|\mathsf{Dir}_{\mathsf{snd}}| = |\mathsf{Dir}_{\mathsf{rcv}}| = 1$. In this case P^G degenerates to a direction-unaware TPS as defined in Section 2.1. If we require $|\mathsf{Dir}_{\mathsf{rcv}}| = 1$, then processes



(a) Bi-directional ring with directions



(b) Directed topology with multiple edges of same direction

Fig. 1. Direction-aware Topologies

cannot choose from which directions to receive the token, but possibly in which direction to send it. Denote the set of all such process templates by $\mathcal{P}_{snd}$. Similarly define $\mathcal{P}_{rcv}$ to be all process templates where $|\mathsf{Dir}_{\mathsf{snd}}| = 1$ — processes cannot choose where to send the token, but possibly from which direction to receive it. Finally, let $\mathcal{P}_{sndrcv}$ be the set of all direction-aware process templates.

Examples. Figure 1(a) shows a bi-directional ring with directions cw (clockwise) and ccw (counterclockwise). Every edge e is labeled with an outgoing direction $dir_{\mathsf{snd}}(e)$ and an incoming direction $dir_{\mathsf{rcv}}(e)$.[2] Using these directions, a process that has the token can choose whether he wants to send it in direction cw or ccw. Depending on its local state, a process waiting for the token can also choose to receive it only from direction cw or ccw.

Figure 1(b) depicts a topology in which process 1 can choose between two outgoing directions. If it sends the token in direction snd_1, it may be received by either process 2 or 3. If however process 2 blocks receiving from direction rcv_1, the token can only be received by process 3. If 3 additionally blocks receiving from rcv_2, then this token-passing transition is disabled.

2.3 Indexed Temporal Logics

Indexed temporal logics (ITL) were introduced in [4,9,8] to model specifications of certain distributed systems. Subsequently one finds a number of variations of indexed temporal-logics in the literature (linear vs. branching, restrictions on the quantification). Thus we introduce Indexed-CTL^* which has these variations (and those in [5]) as syntactic fragments.

Syntactic Fragments of CTL^*, and $\equiv_{\mathsf{TL}}$. We assume the reader is familiar with the syntax and semantics of CTL^*, for a reminder see [2]. For $d \in \mathbb{N}$ let $\mathsf{CTL}^*_d\backslash\mathsf{X}$ denote the syntactic fragment of $\mathsf{CTL}^*\backslash\mathsf{X}$ in which the nesting-depth of path quantifiers is at most d (for a formal definition see [16, Section 4]).

[2] For notational simplicity, we denote both outgoing direction $\mathsf{snd}_{\mathsf{cw}}$ and incoming direction $\mathsf{rcv}_{\mathsf{cw}}$ by cw, and similarly for ccw.

Let TL denote a temporal logic (in this paper these are fragments of $\mathsf{CTL}^*\backslash\mathsf{X}$). For temporal logic TL and LTSs M and N, write $M \equiv_{\mathsf{TL}} N$ to mean that for every formula $\phi \in \mathsf{TL}$, $M \models \phi$ if and only if $N \models \phi$.

Indexed-CTL^*. Fix an infinite set $\mathsf{Vars} = \{x, y, z, \dots\}$ of index variables, i.e., variables with values from $\mathbb{N}$. These variables refer to vertices in the topology.

Syntax. The *Indexed-CTL* formulas over variable set Vars and atomic propositions AP* are formed by adding the following rules to the syntax of CTL^* over atomic propositions $\mathsf{AP} \times \mathsf{Vars}$. We write p_x instead of $(p, x) \in \mathsf{AP} \times \mathsf{Vars}$.

If ϕ is an indexed-CTL^* state (resp. path) formula and $x, y \in \mathsf{Vars}, Y \subset \mathsf{Vars}$, then the following are also indexed-CTL^* state (resp. path) formulas:

- $\forall x.\ \phi$ and $\exists x.\phi$ (i.e., for all/some vertices in the topology, ϕ should hold),
- $\forall x.\ x \in Y \to \phi$ and $\exists x.x \in Y \wedge \phi$ (for all/some vertices that that are designated by variables in Y),
- $\forall x.\ x \in E(y) \to \phi$ and $\exists x.x \in E(y) \wedge \phi$ (for all/some vertices to which there is an edge from the vertex designated by the variable y).

Index Quantifiers. We use the usual shortands (e.g., $\forall x \in Y.\ \phi$ is shorthand for $\forall x.\ x \in Y \to \phi$). The quantifiers introduced above are called called *index quantifiers*, denoted Qx.

Semantics. Indexed temporal logic is interpreted over a system instance P^G (with P a process template and G a topology). The formal semantics are in the full version of the paper [1]. Here we give some examples. The formula $\forall i.\ \mathsf{E}\,\mathsf{F}\, p_i$ states that for every process there exists a path such that, eventually, that process is in a state that satisfies atom p. The formula $\mathsf{E}\,\mathsf{F}\,\forall i.p_i$ states that there is a path such that eventually all processes satisfy atom p simultaneously. We now define the central fragment that includes the former example and not the latter.

Prenex indexed TL and $\{\forall, \exists\}^k$-TL. *Prenex indexed temporal-logic* is a syntactic fragment of indexed temporal-logic in which all quantifiers are at the front of the formula, e.g., prenex indexed $\mathsf{LTL}\backslash\mathsf{X}$ consists of formulas of the form $(Q_1 x_1) \dots (Q_k x_k)\ \varphi$ where φ is an $\mathsf{LTL}\backslash\mathsf{X}$ formula over atoms $\mathsf{AP} \times \{x_1, \dots, x_k\}$, and the $Q_i x_i$s are index quantifiers. Such formulas with k quantifiers will be referred to as *k-indexed*, collectively written $\{\forall, \exists\}^k$-TL. The union of $\{\forall, \exists\}^k$-TL for $k \in \mathbb{N}$ is written $\{\forall, \exists\}^*$-TL and called (full) prenex indexed TL.

2.4 Parameterized Model Checking Problem, Cutoffs, Decidability

A *parameterized topology* $\mathbf{G}$ is a countable set of topologies. E.g., the set of uni-directional rings with all possible initial vertices is a parameterized topology.

$\mathsf{PMCP}_{\mathbf{G}}(-,-)$. The *parameterized model checking problem (PMCP)* for parameterized topology $\mathbf{G}$, processes from $\mathcal{P}$, and parameterized specifications from $\mathcal{F}$, written $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$, is the set of pairs $(\varphi, P) \in \mathcal{F} \times \mathcal{P}$ such that for all $G \in \mathbf{G}$, $P^G \models \varphi$. A *solution* to $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$ is an algorithm that, given a formula $\varphi \in \mathcal{F}$ and a process template $P \in \mathcal{P}$ as input, outputs 'Yes' if for all $G \in \mathbf{G}$, $P^G \models \varphi$, and 'No' otherwise.

Cutoff. A *cutoff* for $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$ is a natural number c such that for every $P \in \mathcal{P}$ and $\varphi \in \mathcal{F}$, the following are equivalent:

– $P^G \models \varphi$ for all $G \in \mathbf{G}$ with $|V_G| \leq c$;

– $P^G \models \varphi$ for all $G \in \mathbf{G}$.

Thus $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$ *does not have a cutoff* iff for every $c \in \mathbb{N}$ there exists $P \in \mathcal{P}$ and $\varphi \in \mathcal{F}$ such that $P^G \models \varphi$ for all $G \in \mathbf{G}$ with $|V_G| \leq c$, and there exists $G \in \mathbf{G}$ such that $P^G \not\models \varphi$.

Observation 1. *If* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$ *has a cutoff, then* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$ *is decidable*

Indeed: if c is a cutoff, let $G_1, \ldots, G_n$ be all topologies G in $\mathbf{G}$ such that $|V_G| \leq c$. The algorithm that solves PMCP takes P, φ as input and checks whether or not $P^{G_i} \models \varphi$ for all $1 \leq i \leq n$.

Note about Decidability. The following statements are not, a priori, equivalent (for given parameterized topology $\mathbf{G}$ and process templates $\mathcal{P}$):

- For every $k \in \mathbb{N}$, $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \{\forall, \exists\}^k\text{-}\mathsf{TL})$ is decidable.
- $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \{\forall, \exists\}^*\text{-}\mathsf{TL})$ is decidable.

The first item says that for every k there *exists* an algorithm A_k that solves the PMCP for k-indexed TL. This does not imply the second item, which says that there exists an algorithm that solves the PMCP for $\cup_{k \in \mathbb{N}} \{\forall, \exists\}^k\text{-}\mathsf{TL}$. If the function $k \mapsto A_k$ is also *computable* (e.g., Theorem 15) then indeed the second item follows: given P, φ, extract the size k of the prenex block of φ, *compute* a description of A_k, and run A_k on P, φ.

For instance, the result of Clarke et al. — that there are cutoffs for k-index $\mathsf{LTL}\backslash\mathsf{X}$ and arbitrary topologies — does not imply that the PMCP for $\{\forall, \exists\}^*\text{-}\mathsf{LTL}\backslash\mathsf{X}$ and arbitrary topologies is decidable. Aware of this fact, the authors state (after Theorem 4) "Note that the theorem does not provide us with an effective means to find the reduction [i.e. algorithm]...".

In fact, we prove (Theorem 14) that there is some parameterized topology such that PMCP is undecidable for prenex indexed $\mathsf{LTL}\backslash\mathsf{X}$.

Existing Results. We restate the known results using our terminology.
A *uni-directional ring* $G = (V, E, x)$ is a topology with $V = [n]$ for some $n \in \mathbb{N}$, there are edges $(i, i+1)$ for $1 \leq i \leq n$ (arithmetic is modulo n), and $x \in V$. Let $\mathbf{R}$ be the parameterized topology consisting of all uni-directional rings.

Theorem 2 (Implict in [8]). *For every* $k \in \mathbb{N}$*, there is a cutoff for the problem* $PMCP_{\mathbf{R}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{CTL}^*\backslash\mathsf{X})$. [3]

Although Clarke et al. [5] do not explicitly state the following theorem, it follows from their proof technique, which we generalize in Section 3.

Theorem 3 (Implicit in [5]). *For every parameterized topology* $\mathbf{G}$*, and every* $k \in \mathbb{N}$*, the problem* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{LTL}\backslash\mathsf{X})$ *has a cutoff.* [4].

[3] The paper explicitly contains the result that 4 is a cutoff for $\forall\forall\text{-}\mathsf{CTL}^*\backslash\mathsf{X}$ on rings. However the proof ideas apply to get the stated theorem.

[4] Khalimov et al. [14, Corollary 2] state that $2k$ is a cutoff if $\mathbf{G}$ is taken to be $\mathbf{R}$. However this is an error: $2k$ is a cutoff only for formulas with no quantifier alternations. See Remark 16 in Section 5.

Theorem 4 ([5, Corollary 3]). *There exists a parameterized topology* $\mathbf{G}$ *and process* $P \in \mathcal{P}_u$ *such that the problem* $\mathsf{PMCP}_{\mathbf{G}}(\{P\}, \{\exists\}^2\text{-}\mathsf{CTL}\backslash\mathsf{X})$ *does not have a cutoff.*

The proof of this theorem defines $\mathbf{G}$, process P, and for every $c \in \mathbb{N}$ a formula φ_c, such that if $G \in \mathbf{G}$ then $P^G \models \varphi_c$ if and only if $|V_G| \leq c$. The formula φ_c is in 2-indexed $\mathsf{CTL}\backslash\mathsf{X}$ and has nesting depth of path quantifiers equal to c.

3 Method for Proving Existence of Cutoffs

We give a method for proving cutoffs for direction-*un*aware TPSs that will be used to prove Theorem 7.

In a k-indexed TL formula $Q_1 x_1 \dots Q_k x_k.\ \varphi$, every valuation of the variables $x_1, \dots, x_k$ designates k nodes of the underlying topology G, say $\bar{g} = g_1, \dots, g_k$. The formula φ can only talk about (the processes in) $\bar{g}$. In order to prove that the PMCP has a cutoff, it is sufficient (as the proof of Theorem 5 will demonstrate) to find conditions on two topologies G, G' and $\bar{g}, \bar{g}'$ that allow one to conclude that P^G and $P^{G'}$ are indistinguishable with respect to φ.

We define two abstractions for a given TPS P^G. The first abstraction simulates P^G, keeping track only of the local states of processes indexed by $\bar{g}$. We call it the *projection* of P^G onto $\bar{g}$.[5] The second abstraction only simulates the movement of the token in G, restricted to $\bar{g}$. We call it the *graph LTS* of G and $\bar{g}$.

Notation. Let $\bar{g}$ denote a tuple $(g_1, \dots, g_k)$ of *distinct* elements of V_G, and $\bar{g}'$ a k-tuple of distinct elements of $V_{G'}$. Write $v \in \bar{g}$ if $v = g_i$ for some i.

The Projection $P^G|\bar{g}$. Informally, the *projection* of P^G onto a tuple of process indices $\bar{g}$ is the LTS P^G and a new labeling that, for every $g_i \in \bar{g}$, replaces the indexed atom p_{g_i} by the atom $p@i$; all other atoms are removed. Thus $p@i$ means that the atom $p \in \mathsf{AP}_{\mathsf{pr}}$ holds in the process with index g_i. In other words, process indices are replaced by their *positions* in $\bar{g}$.

Formally, fix process P, topology G, and k-tuple $\bar{g}$ over V_G. Define the *projection of* $P^G = (S, S_0, \Sigma_{\mathsf{int}} \cup \{\mathsf{tok}\}, \Delta, \Lambda)$ *onto* $\bar{g}$, written $P^G|\bar{g}$ as the LTS $(S, S_0, \Sigma_{\mathsf{int}} \cup \{\mathsf{tok}\}, \Delta, L)$ over atomic propositions $\{p@i : p \in \mathsf{AP}_{\mathsf{pr}}, i \in [k]\}$, where for all $s \in S$ the labeling $L(s)$ is defined as follows: $L(s) := \{p@i : p_{g_i} \in \Lambda(s), i \in [k]\}$.

The Graph LTS $G|\bar{g}$. Informally, $G|\bar{g}$ is an LTS where states are nodes of the graph G, and transitions are edges of G. The restriction to $\bar{g}$ is modeled by labeling a state with the position of the corresponding node in $\bar{g}$.

Let $G = (V, E, x)$ be a topology, and let $\bar{g}$ be a k-tuple over V_G. Define the *graph LTS* $G|g$ as the LTS $(Q, Q_0, \Sigma, \Delta, \Lambda)$ over atomic propositions $\{1, \dots, k\}$, with state set $Q := V$, initial state set $Q_0 := \{x\}$, action set $\Sigma = \{\mathsf{a}\}$, transition relation with $(v, \mathsf{a}, w) \in \Delta$ iff $(v, w) \in E$, and labeling $\Lambda(v) := \{i\}$ if $v = g_i$ for some $1 \leq i \leq k$, and $\emptyset$ otherwise.[6]

[5] Emerson and Namjoshi [8, Section 2.1] define the related notion "LTS projection".

[6] Atomic propositions that have to be true in exactly one state of a structure are called nominals in [3].

Fix a non-indexed temporal logic TL, such as $\mathsf{CTL}^*\backslash\mathsf{X}$. We now define what it means for TL to have the reduction property and the finiteness property. Informally, the reduction property says that if G and G' have the same connectivity (with respect to TL and only viewing k-tuples $\bar{g}, \bar{g}'$) then P^G and $P^{G'}$ are indistinguishable (with respect to TL-formulas over process indices in $\bar{g}, \bar{g}'$).

REDUCTION property for TL[7]

For every $k \in \mathbb{N}$, process $P \in \mathcal{P}_u$, topologies G, G', k-tuples $\bar{g}, \bar{g}'$, if $G|\bar{g} \equiv_{\mathsf{TL}} G'|\bar{g}'$ then $P^G|\bar{g} \equiv_{\mathsf{TL}} P^{G'}|\bar{g}'$.

FINITENESS property for TL

For every $k \in \mathbb{N}$, there are finitely many equivalence classes $[G|\bar{g}]_{\equiv_{\mathsf{TL}}}$ where G is an arbitrary topology, and $\bar{g}$ is a k-tuple over V_G.

Theorem 5 (REDUCTION & FINITENESS $\implies$ Cutoffs for $\{\forall, \exists\}^k$-TL). *If TL satisfies the* REDUCTION *and the* FINITENESS *property, then for every k, $\mathbf{G}$, $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{TL})$ has a cutoff.*

Proof. Fix quantifier prefix $Q_1 x_1 \dots Q_k x_k$. We prove that there exist finitely many topologies $G_1, \cdots, G_N \in \mathbf{G}$ such that for every $G \in \mathbf{G}$ there is an $i \leq N$ such that for all $P \in \mathcal{P}_u$, and all TL-formulas φ over atoms $\mathsf{AP}_{\mathsf{pr}} \times \{x_1, \cdots, x_k\}$

$$P^G \models Q_1 x_1 \dots Q_k x_k.\, \varphi \iff P^{G_i} \models Q_1 x_1 \dots Q_k x_k.\, \varphi$$

In particular, $\max\{|V_{G_i}| : 1 \leq i \leq N\}$ is a cutoff for $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{TL})$.

Suppose for simplicity of exposition that $Q_i x_i$ is a quantifier that also expresses that the value of x_i is different from the values of $x_j \in \{x_1, \dots, x_{i-1}\}$.[8] Fix representatives of $[G|\bar{g}]_{\equiv_{\mathsf{TL}}}$ and define a function r that maps $G|\bar{g}$ to the representative of $[G|\bar{g}]_{\equiv_{\mathsf{TL}}}$. Define a function *rep* that maps $P^G|\bar{g}$ to $P^H|\bar{h}$, where $r(G|\bar{g}) = H|\bar{h}$.

For every $\equiv_{\mathsf{TL}}$-representative $H|\bar{h}$ (i.e., $H|\bar{h} = r(G|\bar{g})$ for some topology G and k-tuple $\bar{g}$), introduce a new Boolean proposition $q_{H|\bar{h}}$. By the FINITENESS property of TL there are finitely many such Boolean propositions, say n.

Define a valuation e_φ (that depends on φ) of these new atoms by

$$e_\varphi(q_{H|\bar{h}}) := \begin{cases} \top & \text{if } P^H|\bar{h} \models \varphi[p_{x_j} \mapsto p@j] \\ \bot & \text{otherwise.} \end{cases}$$

For every $G \in \mathbf{G}$ define Boolean formula $B_G := (\overline{Q_1} g_1 \in V_G) \dots (\overline{Q_k} g_k \in V_G)\; q_{r(G|\bar{g})}$, where $\overline{Q}$ is the Boolean operation corresponding to Q, e.g., $\overline{\exists} g_i \in V_G$ is interpreted as $\bigvee_{g \in V_G \setminus \{g_1, \dots, g_{i-1}\}}$.[9]

[7] Properties of this type are sometimes named *composition* instead of *reduction*, see for instance [15].

[8] All the types of quantifiers defined in Section 2.3, such as $\exists x \in E(y)$, can be dealt with similarly at the cost of notational overhead.

[9] Note that in the Boolean propositions G is fixed while $\bar{g} = (g_1, \dots, g_n)$ ranges over $(V_G)^k$ and is determined by the quantification.

Then (for all P, G and φ)

$$\begin{aligned} & P^G \models Q_1 x_1 \dots Q_k x_k.\ \varphi \\ \iff & Q_1 g_1 \in V_G \dots Q_k g_k \in V_G : P^G \models \varphi[p_{x_j} \mapsto p_{g_j}] \\ \iff & Q_1 g_1 \in V_G \dots Q_k g_k \in V_G : P^G|\bar{g} \models \varphi[p_{x_j} \mapsto p@j] \\ \iff & Q_1 g_1 \in V_G \dots Q_k g_k \in V_G : rep(P^G|\bar{g}) \models \varphi[p_{x_j} \mapsto p@j] \\ \iff & e_\varphi(B_G) = \top \end{aligned}$$

Here $\varphi[p_{x_j} \mapsto p_{g_j}]$ is the formula resulting from replacing every atom in φ of the form p_{x_j} by the atom p_{g_j}, for $p \in \mathsf{AP_{pr}}$ and $1 \leq j \leq k$. Similarly $\varphi[p_{x_j} \mapsto p@j]$ is defined as the formula resulting from replacing (for all $p \in \mathsf{AP_{pr}}, j \leq k$) every atom in φ of the form p_{x_j} by the atom $p@j$. The first equivalence is by the definition of semantics of indexed temporal logic; the second is by the definition of $P^G|\bar{g}$; the third is by the REDUCTION property of TL; the fourth is by the definition of e_φ and *rep*.

Fix $B_{G_1}, \dots, B_{G_N}$ (with $G_i \in \mathbf{G}$) such that every B_G ($G \in \mathbf{G}$) is logically equivalent to some B_{G_i}. Such a finite set of formulas exists since there are 2^{2^n} Boolean formulas (up to logical equivalence) over n Boolean propositions, and thus at most 2^{2^n} amongst the B_G for $G \in \mathbf{G}$.

By the equivalences above conclude that for every $G \in \mathbf{G}$ there exists $i \leq N$ such that $P^G \models Q_1 x_1 \dots Q_k x_k.\ \varphi$ if and only if $P^{G_i} \models Q_1 x_1 \dots Q_k x_k.\ \varphi$. Thus '$\forall G \in \mathbf{G}, P^G \models \varphi$' is equivalent to '$\bigwedge_{i \leq N} e_\varphi(B_{G_i})$' and so the integer $c := \max\{|V_{G_i}| : 1 \leq i \leq N\}$ is a cutoff for $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{TL})$. □

Remark 6. The theorem implies that for every $k, \mathbf{G}$, $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \{\forall, \exists\}^k\text{-}\mathsf{TL})$ is decidable. Further, fix $\mathbf{G}$ and suppose that given k one could compute the finite set $G_1, \cdots, G_N$. Then by the last sentence in the proof one can compute the cutoff c. In this case, $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}, \{\forall, \exists\}^*\text{-}\mathsf{TL})$ is decidable.

4 Existence of Cutoffs for k-indexed $\mathsf{CTL}_d^*\backslash\mathsf{X}$

The following theorem answers Question 1 from the introduction.

Theorem 7. *Let* $\mathbf{G}$ *be a parameterized topology. Then for all* $k, d \in \mathbb{N}$*, the problem* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{CTL}_d^*\backslash\mathsf{X})$ *has a cutoff.*

Corollary 8. *Let* $\mathbf{G}$ *be a parameterized topology. Then for all* $k, d \in \mathbb{N}$*, the problem* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall, \exists\}^k\text{-}\mathsf{CTL}_d^*\backslash\mathsf{X})$ *is decidable.*

To prove the Theorem it is enough, by Theorem 5, to show that the logic $\{\forall, \exists\}^k\text{-}\mathsf{CTL}_d^*\backslash\mathsf{X}$ has the REDUCTION property and the FINITENESS property.

Theorem 9 (Reduction). *For all* $d, k \in \mathbb{N}$*, topologies* G, G'*, processes* $P \in \mathcal{P}_u$*, k-tuples* $\bar{g}$ *over* V_G *and k-tuples* $\bar{g}'$ *over* $V_{G'}$*:*

If $G|\bar{g} \equiv_{\mathsf{CTL}_d^*\backslash\mathsf{X}} G'|\bar{g}'$ *then* $P^G|\bar{g} \equiv_{\mathsf{CTL}_d^*\backslash\mathsf{X}} P^{G'}|\bar{g}'$.

The idea behind the proof is to show that paths in P^G can be simulated by paths in $P^{G'}$ (and vice versa). Given a path π in P^G, first project it onto G to get a path ρ that records the movement of the token, then take an equivalent path ρ' in G' which exists since $G|\bar{g} \equiv_{\mathsf{CTL}^*_d\backslash\mathsf{X}} G'|\bar{g}'$, and then lift ρ' up to get a path π' in $P^{G'}$ that is equivalent to π. This lifting step uses the assumption that process P is in $\mathcal{P}_u$, i.e., P cannot control where it sends the token, or from where it receives it. The proof can be found in the full version of the paper [1].

Remark 10. As immediate corollaries we get that the REDUCTION property holds with $\mathsf{TL} = \mathsf{LTL}\backslash\mathsf{X}$ (take $d = 1$), $\mathsf{CTL}^*\backslash\mathsf{X}$ (since if the assumption holds with $\mathsf{TL} = \mathsf{CTL}^*\backslash\mathsf{X}$ then the conclusion holds with $\mathsf{TL} = \mathsf{CTL}^*_d\backslash\mathsf{X}$ for all $d \in \mathbb{N}$, and thus also for $\mathsf{TL} = \mathsf{CTL}^*\backslash\mathsf{X}$) and, if P is finite, also for $\mathsf{TL} = \mathsf{CTL}\backslash\mathsf{X}$ (since $\mathsf{CTL}\backslash\mathsf{X}$ and $\mathsf{CTL}^*\backslash\mathsf{X}$ agree on finite structures).

Finiteness Theorem. Theorem 4 ([5, Corollary 3]) states that there exists $\mathbf{G}$ such that the problem $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \exists\exists\text{-}\mathsf{CTL}^*\backslash\mathsf{X})$ does not have a cutoff. We observed that the formulas from their result have unbounded nesting depth of path quantifiers. This leads to the idea of stratifying $\mathsf{CTL}^*\backslash\mathsf{X}$ by nesting depth.

Recall from Section 2.3 that i) $\mathsf{CTL}^*_d\backslash\mathsf{X}$ denotes the syntactic fragment of $\mathsf{CTL}^*\backslash\mathsf{X}$ in which formulas have path-quantifier nesting depth at most d; ii) $M \equiv_{\mathsf{CTL}^*_d\backslash\mathsf{X}} N$ iff M and N agree on all $\mathsf{CTL}^*_d\backslash\mathsf{X}$ formulas. Write $[M]_{\mathsf{CTL}^*_d\backslash\mathsf{X}}$ for the set of all LTSs N such that $M \equiv_{\mathsf{CTL}^*_d\backslash\mathsf{X}} N$.

Following the method of Section 3 we prove that the following FINITENESS property holds (where k represents the number of process-index quantifiers in the prenex indexed temporal logic formula).

Remark 11. For ease of exposition we sketch a proof under the assumption that path quantifiers in formulas ignore runs in which the token does not visit every process infinitely often. This is an explicit restriction in [5] and implicit in [8]. In the full version [1] we remove this restriction. For the purpose of this paper this restriction only affects the explicit cutoffs in Theorem 15.

Theorem 12 (Finiteness). *For all positive integers k and d, there are finitely many equivalence classes $[G|\bar{g}]_{\equiv_{CTL^*_d\backslash X}}$ where G is an arbitrary topology, and $\bar{g}$ is a k-tuple over V_G.*

Proof Idea. We provide an algorithm that given positive integers k, d, topology G, k-tuple $\bar{g}$ over V_G, returns a LTS $\textsc{con}_d G|\bar{g}$ such that $G|\bar{g} \equiv_{\mathsf{CTL}^*_d\backslash\mathsf{X}} \textsc{con}_d G|\bar{g}$. Moreover, we prove that for fixed k, d the range of $\textsc{con}_d\text{\_}|\text{\_}$ is finite.

Recursively define a marking function μ_d that associates with each $v \in V_G$ a finite set (of finite strings over alphabet $\mu_{d-1}(V_G)$). For the base case define $\mu_0(v) := \Lambda(v)$, the labeling of $G|\bar{g}$. The marking $\mu_d(v)$ stores (representatives) of all strings of μ_{d-1}-labels of paths that start in v and reach some element in $\bar{g}$. The idea is that $\mu_d(v)$ determines the set of $\mathsf{CTL}^*_d\backslash\mathsf{X}$ formulas that hold in $G|\bar{g}$ with initial vertex v, as follows: stitch together these strings, using elements of $\bar{g}$ as stitching points, to get the $\mathsf{CTL}^*_d\backslash\mathsf{X}$ types of the infinite paths starting in

v. This allows us to define a topology, called the d-contraction $\textsc{con}_d G|\bar{g}$, whose vertices are the μ_d-markings of vertices in G. In the full version [1] we prove that $G|\bar{g}$ is $\mathsf{CTL}^*_d\backslash\mathsf{X}$-equivalent to its d-contraction, and that the number of different d-contractions is finite, and depends on k and d.

Definition of d-contraction $\textsc{con}_d G|\bar{g}$. Next we define d-contractions.

Marking μ_d. Fix $k, d \in \mathbb{N}$, topology G, and k-tuple $\bar{g}$ over V_G. Let Λ be the labeling-function of $G|\bar{g}$, i.e., $\Lambda(v) = \{i\}$ if $v = g_i$, and $\Lambda(v) = \emptyset$ for $v \notin \bar{g}$. For every vertex $v \in V_G$ define a set $X(v)$ of paths of G as follows: a path $\pi = \pi_1 \dots \pi_t$, say of length t, is in $X(v)$ if π starts in v, none of $\pi_1, \dots, \pi_{t-1}$ is in $\bar{g}$, and $\pi_t \in \bar{g}$. Note that $X(g_i) = \{g_i\}$.

Define the marking μ_d inductively:

$$\mu_d(v) := \begin{cases} \Lambda(v) & \text{if } d = 0 \\ \{\mathsf{destutter}(\mu_{d-1}(\pi_1) \dots \mu_{d-1}(\pi_t)) : \pi_1 \dots \pi_t \in X(v), t \in \mathbb{N}\} & \text{if } d > 0, \end{cases}$$

where $\mathsf{destutter}(w)$ is the maximal substring s of w such that for every two consecutive letters s_i and s_{i+1} we have that $s_i \neq s_{i+1}$. Informally, remove identical consecutive letters of w to get the 'destuttering' $\mathsf{destutter}(w)$.

The elements of $\mu_d(v)$ ($d > 0$) are finite strings over the alphabet $\mu_{d-1}(V_G)$. For instance, strings in $\mu_1(v)$ are over the alphabet $\{\{1\}, \{2\}, \dots, \{k\}, \emptyset\}$.

Equivalence relation $\sim_d$. Vertices $v, u \in V_G$ are *d-equivalent*, written $u \sim_d v$, if $\mu_d(v) = \mu_d(u)$. We say that $\sim_d$ *refines* $\sim_j$ if $u \sim_d v$ implies $u \sim_j v$.

Lemma 13. *If $0 \leq j < d$, then $\sim_d$ refines $\sim_j$.*

Indeed, observe that for all nodes v, all strings in $\mu_d(v)$ start with the letter $\mu_{d-1}(v)$. Thus $\mu_d(v) = \mu_d(u)$ implies that $\mu_{d-1}(v) = \mu_{d-1}(u)$. In other words, if $u \sim_d v$ then $u \sim_{d-1} v$, and thus also $u \sim_j v$ for $0 \leq j < d$.

d-contraction $\textsc{con}_d G|\bar{g}$. Define an LTS $\textsc{con}_d G|\bar{g}$ called the d-contraction of $G|\bar{g}$ as follows. The nodes of the contraction are the $\sim_d$-equivalence classes. Put an edge between $[u]_{\sim_d}$ and $[v]_{\sim_d}$ if there exists $u' \in [u]_{\sim_d}, v' \in [v]_{\sim_d}$ and an edge in G from u' to v'. The initial state is $[x]_{\sim_d}$ where x is the initial vertex of G. The label of $[u]_{\sim_d}$ is defined to be $\Lambda(u)$ — this is well-defined because, by Lemma 13, $\sim_d$ refines $\sim_0$.

In the full version [1] we prove that $G|\bar{g}$ is $\mathsf{CTL}^*_d\backslash\mathsf{X}$-equivalent to its d-contraction, and that the number of different d-contractions is finite, and depends on k and d. $\square$

5 Cutoffs for k-index $\mathsf{CTL}^*\backslash\mathsf{X}$ and Concrete Topologies

The following two theorems answer Question 2 from the introduction, regarding the PMCP for specifications from $\{\forall, \exists\}^*$-$\mathsf{CTL}^*\backslash\mathsf{X}$.

First, the PMCP is undecidable for certain (pathological) parameterized topologies $\mathbf{G}$ and specifications from $\{\forall, \exists\}^*$-$\mathsf{CTL}^*\backslash\mathsf{X}$.

Theorem 14. *There exists a process $P \in \mathcal{P}_u$, and parameterized topologies* $\mathbf{G}$, $\mathbf{H}$, *such that the following PMCPs are undecidable*
1. $\mathsf{PMCP}_{\mathbf{G}}(\{P\}, \{\forall, \exists\}^*\text{-}\mathsf{LTL}\backslash\mathsf{X})$.
2. $\mathsf{PMCP}_{\mathbf{H}}(\{P\}, \{\forall, \exists\}^2\text{-}\mathsf{CTL}\backslash\mathsf{X})$.

Moreover, $\mathbf{G}$ *and* $\mathbf{H}$ *can be chosen to be computable sets of topologies.*

Second, PMCP is decidable for certain (regular) parameterized topologies and specifications from $\{\forall\}^*$-$\mathsf{CTL}^*\backslash\mathsf{X}$. This generalizes results from Emerson and Namjoshi [8] who show this result for $\{\forall\}^k$-$\mathsf{CTL}^*\backslash\mathsf{X}$ with $k = 1, 2$ and uni-directional ring topologies. By Remark 11, these cutoffs apply under the assumption that we ignore runs that do not visit every process infinitely often.

Theorem 15. *If* $\mathbf{G}$ *is as stated, then* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall\}^k\text{-}\mathsf{CTL}^*\backslash\mathsf{X})$ *has the stated cutoff.*
1. *If* $\mathbf{G}$ *is the set of uni-directional rings, then* $2k$ *is a cutoff.*
2. *If* $\mathbf{G}$ *is the set of bi-directional rings, then* $2k$ *is a cutoff.*
3. *If* $\mathbf{G}$ *is the set of cliques, then* $k+1$ *is a cutoff.*
4. *If* $\mathbf{G}$ *is the set of stars, then* $k+1$ *is a cutoff.*

Consequently, for each $\mathbf{G}$ *listed,* $\mathsf{PMCP}_{\mathbf{G}}(\mathcal{P}_u, \{\forall\}^*\text{-}\mathsf{CTL}^*\backslash\mathsf{X})$ *is decidable.*

This theorem is proved following Remark 6: given k, d, we compute a set $G_1, \dots, G_N \in \mathbf{G}$ such that every B_G for $G \in \mathbf{G}$ is logically equivalent to some B_{G_i}, where the Boolean formula B_G is defined as $\bigwedge_{\bar{g}} q_{\mathrm{CON}_d G|\bar{g}}$. To do this, note that B_G is logically equivalent to B_H if and only if $\{\mathrm{CON}_d G|\bar{g} : \bar{g} \in V_G\} = \{\mathrm{CON}_d H|\bar{h} : \bar{h} \in V_H\}$ (this is where we use that there is no quantifier alternation). So it is sufficient to prove that, if c is the stated cutoff,

$$|V_G|, |V_H| \geq c \implies \{\mathrm{CON}_d G|\bar{g} : \bar{g} \in V_G\} = \{\mathrm{CON}_d H|\bar{h} : \bar{h} \in V_H\}$$

To illustrate how to do this, we analyze the case of uni-directional rings and cliques (the other cases are similar).

Uni-directional rings. Suppose $\mathbf{G}$ are the uni-directional rings and let $G \in \mathbf{G}$. Fix a k-tuple of distinct elements of V_G, say $(g_1, g_2, \dots, g_k)$. Define a function $f : V_G \to \{g_1, \dots, g_k\}$ that maps v to the first element of $\bar{g}$ on the path $v, v+1, v+2, \dots$ (addition is mod $|V_G|$). In particular $f(g_i) = g_i$ for $i \in [k]$. In the terminology of the proof of Theorem 12, $X(v)$ consists of the simple path $v, v+1, \cdots, f(v)$.

We now describe μ_d. Clearly $\mu_d(g_i) = \{\mu_{d-1}(g_i)\}$. By induction on d one can prove that if $v \notin \bar{g}$ with $f(v) = g_j$ then $\mu_d(v) = \{\mu_{d-1}(v) \cdot \mu_{d-1}(g_j)\}$. So for every $d > 1$, the equivalences $\sim_d$ and $\sim_1$ coincide.

d	0	1	2	…
$\mu_d(v)$ for $v = g_i$	$\{i\}$	$\{\{i\}\}$	$\{\{\{i\}\}\}$	…
$\mu_d(v)$ if $v \notin \bar{g}$ and $f(v) = g_j$	$\emptyset$	$\{\emptyset \cdot \{j\}\}$	$\{\{\emptyset \cdot \{j\}\} \cdot \{\{j\}\}\}$	…

Thus for every $k \in \mathbb{N}$, the d-contraction $\mathrm{CON}_d G|\bar{g}$ is a ring of size at most $2k$ (in particular, it is independent of d). In words, the d-contraction of G is the ring resulting by identifying adjacent elements not in $\bar{g}$. It is not hard to see that if G, H are rings such that $|V_G|, |V_H| \geq 2k$ then for every $\bar{g}$ there exists $\bar{h}$ such that $\text{CON}_d G|\bar{g} = \text{CON}_d H|\bar{h}$.

Cliques. Fix $n \in \mathbb{N}$. Let G be a clique of size n. That is: $V_G = [n]$ and $(i, j) \in E_G$ for $1 \leq i \neq j \leq n$. Fix a k-tuple of distinct elements of V_G, say $(g_1, g_2, \dots, g_k)$. We now describe $\mu_d(v)$. Clearly $\mu_d(g_i) = \{\mu_{d-1}(g_i)\}$ and for $v \notin \bar{g}$ we have $\mu_d(v) = \{\mu_{d-1}(v) \cdot \mu_{d-1}(j) : j \in [k]\}$. So for every $d > 1$, the equivalences $\sim_d$ and $\sim_1$ coincide, and the d-contraction $\text{CON}_d G|\bar{g}$ is the clique of size $k + 1$. In words, the d-contraction of G results from G by identifying all vertices not in $\bar{g}$. It is not hard to see that if G, H are cliques such that $|V_G|, |V_H| \geq k + 1$ then for every $\bar{g}$ (of size k) there exists $\bar{h}$ such that $\text{CON}_d G|\bar{g} = \text{CON}_d H|\bar{h}$.

Remark 16. For cliques and stars, $k+1$ is also a cutoff for $\{\forall, \exists\}^k$-$\mathsf{CTL}^*\backslash\mathsf{X}$. Also, $2k$ is *not* a cutoff for uni-rings and $\{\forall, \exists\}^k$-$\mathsf{LTL}\backslash\mathsf{X}$ as stated in [14, Corollary 2]. To see this, let tok_i express that the process with index i has the token, and $adj(k, i) := tok_i \to tok_i \mathsf{U}\, tok_k \vee tok_k \to tok_k \mathsf{U}\, tok_i$. Then the formula $\exists i \exists j \forall k.\ adj(k, i) \vee adj(k, j)$, holds in the ring of size 6, but not 7.

6 There Are No Cutoffs for Direction-Aware Systems

In the following, we consider systems where processes can choose which directions are used to send or receive the token, i.e., process templates are from $\mathcal{P}_{snd}$, $\mathcal{P}_{rcv}$, or $\mathcal{P}_{sndrcv}$. Let $\mathbf{B}$ be the parameterized topology of all bi-directional rings, with directions cw (clockwise) and ccw (counter-clockwise). The following theorem answers Question 3 from the introduction.

Theorem 17. *1. $PMCP_{\mathbf{B}}(\mathcal{P}_{sndrcv}, \forall$-$LTL\backslash X)$ is undecidable.*
2. For $\mathcal{F}$ equal to $\{\forall\}^9$-$LTL\backslash X$ or $\{\exists\}^9$-$CTL\backslash X$, and $\mathcal{P} \in \{\mathcal{P}_{snd}, \mathcal{P}_{rcv}\}$, there exists a parameterized topology $\mathbf{G}$ such that $PMCP_{\mathbf{G}}(\mathcal{P}, \mathcal{F})$ is undecidable.

Proof Idea. We reduce the non-halting problem of two-counter machines (2CMs) to the PMCP. The idea is that one process, the *controller*, simulates the finite-state control of the 2CM. The other processes, arranged in a chain or a ring, are *memory processes*, collectively storing the counter values with a fixed memory per process. This allows a given system to simulate a 2CM with bounded counters. Since a 2CM terminates if and only if it terminates for some bound on the counter values, we can reduce the non-halting problem of 2CMs to the PMCP. The main work is to show that the controller can issue commands, such as 'increment counter 1' and 'test counter 1 for zero'. We give a detailed proof sketch for part 1 of the theorem, and then outline a proof for part 2.
1. $\forall$-$LTL\backslash X$ and $\mathcal{P}_{sndrcv}$ in bi-directional rings.

The process starting with the token becomes the controller, all others are memory, each storing one bit for each counter of the 2CM. The current value of a counter c is the total number of corresponding bits (c-bits) set to 1. Thus, a system with n processes can store counter values up to $n - 1$.

Fix a numbering of 2CM-commands, say $0 \mapsto$ 'increment counter 1', $1 \mapsto$ 'decrement counter 1', $2 \mapsto$ 'test counter 1 for zero', etc. Every process has a command variable that represents the command to be executed when it receives the token from direction ccw.

If the controller sends the token in direction cw, the memory processes will increment (mod 6) the command variable, allowing the controller to encode which command should be executed. Every process just continues to pass the token in direction cw, until it reaches the controller again.

If the controller sends the token in direction ccw, then the memory processes try to execute the command currently stored. If it is an 'increment counter c' or 'decrement counter c' command, the memory process tries to execute it (by incrementing/decrementing its c-bit). If the process cannot execute the command (because the c-bit is already 1 for an increment, or 0 for a decrement), then it passes the token along direction ccw and remembers that a command is being tried. If the token reaches a memory process which can execute the command, then it does so and passes the token back in direction cw. The processes that remembered that a command is being tried will receive the token from direction cw, and know that the command has been successfully executed, and so will the controller. If the controller gets the token from ccw, the command failed. In this case, the controller enters a loop in which it just passes the token in direction cw (and no more commands are executed).

If the command stored in the memory processes is a 'test for zero counter c', then the processes check if their c-bit is 0. If this is the case, it (remembers that a command is being tried and) passes the token to the next process in direction ccw. If the token reaches a process for which the c-bit is 1, then this process sends the token back in direction cw. Other memory processes receiving it from cw (and remembering that the command is being tried), pass it on in direction cw. In this case, the controller will receive the token from cw and know that counter c is not zero. On the other hand, if all memory processes store 0 in their c-bit, then they all send the token in direction ccw. Thus, the controller will receive it from ccw and knows that counter c currently is zero. To terminate the command, it sends the token in direction cw, and all processes (which remembered that a command is being tried), know that execution of this command is finished.

With the description above, a system with $n-1$ memory processes can simulate a 2CM as long as counter values are less than n. Let $HALT$ be an atomic proposition that holds only in the controller's halting states. Then solving the PMCP for $\forall i\, \mathsf{G}\, \neg HALT_i$ amounts to solving the non-halting problem of the 2CM.

2. $\{\forall\}^9$*-LTL\X and* $\mathcal{P}_{snd}$*.*

We give a proof outline. In this case there are $2n$ memory processes, n for each counter $c \in \{1,2\}$. The remaining 9 processes are special and called 'controller', 'counter c is zero', 'counter c is not zero', 'counter c was incremented', and 'counter c was decremented'. When the controller wants to increment or decrement counter c, it sends the token non-deterministically to some memory process for counter c. When the controller wants to test counter c for zero, it sends the token to the first memory process. When a memory process receives the token it does not know who sent it, and in particular does not know the intended command. Thus, it non-deterministically takes the appropriate action for one of the possible commands. If its bit is set to 0 then it either i) increments its bit and sends the token to a special process 'counter c was incremented', or ii) it sends the token to the next memory node in the chain, or to the special process 'counter c is zero' if it is the last in the chain. If its bit is set to 1 then it either i) decrements its bit and sends the token to a special process 'counter c was decremented', or ii) sends the token to a special process 'counter c is not zero'.

Even though incoming directions are not available to the processes, we can write the specification such that, out of all the possible non-deterministic runs, we only consider those in which the controller receives the token from the expected special node (the formula requires one quantified index variable for each of the special nodes). So, if the controller wanted to increment counter c it needs to receive the token from process 'counter c was incremented'. If the controller receives the token from a different node, it means that a command was issued but not executed correctly, and the formula disregards this run. Otherwise, the system of size $2n+1$ correctly simulates the 2CM until one of the counter values exceeds n. $\square$

7 Extensions

There are a number of extensions of direction-unaware TPSs for which the theorems that state existence of cutoffs (Theorems 7 and 15) still hold. We describe these in order to highlight assumptions that make the proofs work:

1. Processes can be infinite-state.
2. The EN-restriction on the process template P can be relaxed: replace item *vii)* in Definition 2.1 by "For every state q that has the token there is a finite path $q \dots q'$ such that q' does not have the token, and for every q that does not have the token there is a finite path $q \dots q'$ such that q' has the token".
3. One can further allow *direction-sensing* TPSs, which is a direction-aware TPS with an additional restriction on the process template: "If $q \xrightarrow{\mathsf{d}} q' \in \delta$ for some direction $d \in \mathsf{Dir}_{\mathsf{snd}}$, then for every $d \in \mathsf{Dir}_{\mathsf{snd}}$ there exists a transition $q \xrightarrow{\mathsf{d}} q'' \in \delta$"; and a similar statement for $\mathsf{Dir}_{\mathsf{rcv}}$. Informally: we can allow processes to change state according to the direction that the token is (non-deterministically) sent to or received, but the processes are not allowed to block any particular direction.
4. One can further allow the token to carry a value but with the strong restriction that from every state that has the token and every value v there is a path of internal actions in P which eventually sends the token with value v, and the same for receiving.

These conditions on P all have the same flavor: they ensure that a process can not choose what information to send/receive, whether that information is a value on the token or a direction for the token.

8 Related Work

Besides the results that this paper is directly based on [17,8,5], there are several other relevant papers.

Emerson and Kahlon [7] consider token-passing in uni- and bi-directional rings, where processes are direction-aware and tokens carry messages (but can only be changed a bounded number of times). However, the provided cutoff theorems only hold for specifications that talk about two processes (in a uni-directional ring) or one process (in a bi-directional ring), process templates need to be deterministic, an cutoffs depend on the size of the process implementation.

German and Sistla [12] provide cutoffs for the PMCP for systems with pairwise synchronization. Although pairwise synchronization can simulate token-passing, their cutoff results are restricted to cliques and 1-indexed LTL. Moreover, their proof uses vector-addition systems with states and their cutoff depends on the process template and the specification formula.

Delzanno et al. [6] study a model of broadcast protocols on arbitrary topologies, in which a process can synchronize with all of its available neighbors 'at once' by broadcasting a message (from a finite set of messages). They prove undecidability of PMCP for systems with arbitrary topologies and 1-indexed safety properties, and that the problem becomes decidable if one restricts the topologies to 'graphs with bounded paths' (such as stars). Their proof uses the machinery of well-structured transitions systems, and no cutoffs are provided. They also show undecidability of the PMCP in the case of non-prenex indexed properties of the form $\mathsf{G}(\exists i.s(i) \in B)$ on general and the restricted topologies.

Rabinovich [15, Section 4] proves, using the composition method, that if monadic second-order theory of the set of topologies in $\mathbf{G}$ is decidable, then the PMCP is decidable for propositional modal logic. The systems considered are defined by a very general notion of product of systems (which includes our token passing systems as a subcase).

The PMCP for various fragments of non-prenex indexed LTL is undecidable, see German and Sistla [12, Section 6] for systems with pairwise synchronization, and John et al. [13, Appendix A] for systems with no synchronization at all.

9 Summary

The goal of this work was to find out under what conditions there are cutoffs for temporal logics and token-passing systems on general topologies. We found that stratifying prenex indexed $\mathsf{CTL}^*\backslash\mathsf{X}$ by nesting-depth of path quantifiers allowed us to recover the existence of cutoffs; but that there are no cutoffs if the processes are allowed to choose the direction of the token. In all the considered cases where there is no cutoff we show that the PMCP problem is actually undecidable.

Our positive results are provided by a construction that generalizes and unifies the known positive results, and clearly decomposes the problem into two aspects: tracking the movement of the token through the underlying topology, and simulating the internal states of the processes that the specification formula can see. The construction yields small cutoffs for common topologies (such as rings, stars, and cliques) and specifications from prenex indexed $\mathsf{CTL}^*\backslash\mathsf{X}$.

Acknowledgments. We thank Roderick Bloem for detailed comments on numerous drafts and Krishnendu Chatterjee for important comments regarding the structure of the paper. We thank Roderick Bloem, Igor Konnov, Helmut Veith, and Josef Widder for discussions at an early stage of this work that, in particular, pointed out the relevance of direction-unawareness in [5].

References

1. Aminof, B., Jacobs, S., Khalimov, A., Rubin, S.: Parameterized Model Checking of Token-Passing Systems, pre-print on arxiv.org (2013)
2. Baier, C., Katoen, J.P., et al.: Principles of model checking, vol. 26202649. MIT Press, Cambridge (2008)
3. Bonatti, P.A., Lutz, C., Murano, A., Vardi, M.Y.: The complexity of enriched μ-calculi. In: Logical Methods in Computer Science (LMCS 2008), vol. 4(3:11), pp. 1–27 (2008)
4. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. 81, 13–31 (1989)
5. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
6. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
7. Emerson, E.A., Kahlon, V.: Parameterized model checking of ring-based message passing systems. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 325–339. Springer, Heidelberg (2004)
8. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003)
9. 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)
10. Emerson, E.A., Namjoshi, K.: Reasoning about rings. In: POPL, pp. 85–94 (1995)
11. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Symposium on Logic in Computer Science, p. 352 (1999)
12. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
13. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Counter attack on byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms. CoRR abs/1210.3846 (2012)
14. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 108–127. Springer, Heidelberg (2013)
15. Rabinovich, A.: On compositionality and its limitations. ACM Trans. Comput. Logic 8(1) (January 2007)
16. Shamir, S., Kupferman, O., Shamir, E.: Branching-depth hierarchies. ENTCS 39(1), 65–78 (2003)
17. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. 28(4), 213–214 (1988)

Modularly Combining Numeric Abstract Domains with Points-to Analysis, and a Scalable Static Numeric Analyzer for Java

Zhoulai Fu[⋆]

Université de Rennes 1 – INRIA, France

Abstract. This paper contributes to a new abstract domain that combines static numeric analysis and points-to analysis. One particularity of this abstract domain lies in its high degree of modularity, in the sense that the domain is constructed by reusing its combined components as black-boxes. This modularity dramatically eases the proof of its soundness and renders its algorithm intuitive. We have prototyped the abstract domain for analyzing real-world Java programs. Our experimental results show a tangible precision enhancement compared to what is possible by traditional static numeric analysis, and this at a cost that is comparable to the cost of running the numeric and pointer analyses separately.

1 Introduction

Static numeric analysis – that approximates values of scalar variables and their relationship – has drawn on a rich body of techniques including abstract domains of intervals [9], polyhedron [13] and octagons [24] etc. which have found their way into mature implementations. In a similar way, the analysis of properties describing the shape of data structures in the heap has flourished into a rich set of points-to and alias analyses which also have provided a range of production-quality analyzers. However, when extending numeric analyses to heap-manipulating programs we are immediately faced with the issues that pointers introduce *aliases* which make program reasoning difficult because understanding the communication between numeric properties and dynamic data structures is needed. This gives rise to *the problem of combining static numeric analysis and heap analysis.*

The combination of the two analyses has been studied, but the solutions proposed so far tend to be complex to implement or impractical to analyze large programs. For example, Simon [27] shows how to combine *ad hoc* numeric abstract domains with manually refined flow-sensitive points-to analyses. His combination approach requires extensive experiences and intimate familiarity with the abstract domains themselves, thereby hard to implement. Miné's abstraction [23], by contrast, is designed to be modular. The purpose was to lift

⋆ The research leading to these results has received financial help from AX – L'Association des Anciens Élèves et Diplômés de l'École polytechnique. 5, rue Descartes 75005 PARIS.

existing abstract domains in ASTREE [3] developed with several man-years to cope with pointer-aware programs. Reusing existing components as modules is particularly important in that context. However, Miné's framework is based on type-based pointer analysis, which is cheap but too coarse by its nature. This prohibits the general practicability of the Miné's analysis. At the other extreme, shape-analysis [26] based approaches come with sophisticated pointer analyses and can indeed infer non-trivial properties. However, analyses that are based on shape abstraction can hardly (see [5,32] for exceptions) run on large programs.

Different from the work mentioned above, our objective is to develop a combined analysis satisfying the following requirements:

- **Modular design:** The combined analysis should enable the reusing of existing analyses that have been developed since decades. The construction of the combined analysis should only depend on the interfaces, not the specific implementations, of its components.
- **Scalability:** We are seeking a tool that runs on codes of hundreds of thousands of lines. We examine the feasibility of our analysis over moderate and large sized benchmarks, and ensure that the combined analysis only presents small complexity overhead compared with its component analyses.
- **Precision:** Although the query of scalability inevitably demands a sacrifice on precision, we inspect that the combined analysis has to be, at least, as precise as its components.

The core contribution of this work is a theoretical foundation that combines in a generic manner

- an abstract domain dedicated to static numeric analysis of programs without allocations, and
- an abstract domain for points-to static analysis.

On the practical side, we have implemented the abstract domain, using the Java Optimization Framework SOOT [29] as the front-end, and relying on the abstract domains from existing static analysis libraries such as the Parma Polyhedra Library PPL [1] and the SOOT Pointer Analysis Research Kit SPARK [20]. This prototype analyzer, called NumP, has been run on *all* 11 programs in the Dacapo-2006-MR2 [4] benchmark suite. The suite is composed of moderate and large sized program with rich object behaviors and demanding memory system requirements. Our experiments confirm that the combined analysis is feasible even for large-sized programs and that it discovers significantly more program properties than what is possible by pure numeric analysis, and this at a cost that is comparable to the cost of running the numeric and pointer analysis separately.

1.1 Organization of the Paper

The interfaces of traditional numeric and pointer analyses are specified in Sect. 2. The intuition of our analysis is illustrated with a small example in Sect. 3. In Sect. 4, we define the modeled language and its concrete semantics. The abstract domain and its operators are presented in Sect. 5. Experimental results are shown in Sect. 6. Finally, we compare our analysis with related work and conclude in Sect. 7 and 8.

The formal underpinning and semantic correctness of the combination technique are presented in the author's Ph.D. thesis [17].

2 Analysis Interfaces

This section is define the interfaces of two existing analysis, static numeric analysis and points-to analysis.

General notation. For a given set U, the notation $U_\perp$ means the disjoint union $U \cup \{\perp\}$. Given a mapping $m \in A \to B_\perp$, we express the fact that m is undefined in a point x by $m(x) = \perp$.

Syntactical notations. Primary data types include: scalar numbers in $\mathbb{I}$, where $\mathbb{I}$ can be integers, rationales or reals; and references (or pointers) in *Ref*. Primary syntactical entities include the universe of *local variables* and *fields.* They are denoted by *Var* and *Fld* respectively. An *access path* is either a variable or a variable followed by a sequence of fields. The universe of access paths is denoted by *Path*. We subscript Var_τ, Field_τ, or Path_τ with $\tau \in \{n, p\}$ to indicate their types as a scalar number or a reference, respectively. The elements in these sets can also be sub-scripted with types. The types will be omitted if they are clear from context.

We use Imp_n to refer to the basic statements only involving numeric variables and use the meta-variables s_n to range over these statements. Similarly, we let Imp_p be the statements that only use pointer variables and let s_p range over these statements. Below we list the syntactical entities and meta-variables used to range over them.

$k \in \mathbb{I}$	scalar numbers
$r \in \mathit{Ref}$	concrete references
$x_\tau, y_\tau \in \text{Var}_\tau$	numeric/pointer variables
$f_\tau, g_\tau \in \text{Field}_\tau$	numeric/pointer fields
$\boldsymbol{u}_\tau, \boldsymbol{v}_\tau \in \mathit{Path}_\tau$	numeric/pointer access paths
$s_n \in \mathsf{Imp}_n$	$x_n = k \mid x_n = y_n \mid x_n = y_n \diamond z_n \mid x_n \bowtie y_n$
$s_p \in \mathsf{Imp}_p$	$x_p = \texttt{new} \mid x_p = y_p.f_p \mid x_p = y_p \mid x_p.f_p = y_p$

where $\diamond \in \{+, -, *, /\}$, and $\bowtie$ is an arithmetic comparison operator.

2.1 Static Numeric Analysis

Static numeric analysis can be modeled as an abstract interpretation of Imp_n.

We use the term *numeric property* [22] for any conjunction of formula in a certain theory of arithmetic. For example, the numeric property $\{x^2 + y^2 \leq 1, \quad x \leq 0, \quad y \leq 0\}$ is composed of the conjunction of three arithmetic formulas

As usual, an *environment* maps variables to their values. We consider *numeric environments*:

$$Num \triangleq Var_n \to \mathbb{I}_\perp \tag{1}$$

The relationship between an environment and a property can be formalized by the concept of *valuation*. We say that n is a valuation of $\mathsf{n}^\sharp$, denoted by

$$\mathsf{n} \models \mathsf{n}^\sharp \tag{2}$$

if $\mathsf{n}^\sharp$ becomes a tautology after each of its free variables, if any, has been replaced by its corresponded value in n.

Definition 1 (Interface of the traditional numeric analyzer)

$$(\mathsf{Imp}_n, \wp(Num), [\![\cdot]\!]_n^\natural, \gamma_n, Num^\sharp, [\![\cdot]\!]_n^\sharp)$$

The concrete numeric domain *and the* abstract numeric domain *for the language* Imp_n *are* $\wp(Num)$ *and* $Num^\sharp$ *respectively. They are related by the* concretization function $\gamma_n : Num^\sharp \to \wp(Num)$ *defined by* $\gamma_n(\mathsf{n}^\sharp) = \{n \in Num \mid n \models \mathsf{n}^\sharp\}$.

The partial order $\sqsubseteq$ *is consistent with the monotonicity of* γ_n, i.e., $\mathsf{n}_1^\sharp \sqsubseteq \mathsf{n}_2^\sharp$ *implies* $\gamma_n(\mathsf{n}_1^\sharp) \subseteq \gamma_n(\mathsf{n}_2^\sharp)$. *For each statement* s_n *of* Imp_n*, the concrete semantics is given by a standard transfer function* $[\![s_n]\!]_n^\natural \in \wp(Num) \to \wp(Num)$. *The abstract semantics* $[\![\cdot]\!]_n^\sharp$ *satisfies the soundness condition:*

$$[\![\cdot]\!]_n^\natural \circ \gamma_n \subseteq \gamma_n \circ [\![\cdot]\!]_n^\sharp \tag{3}$$

At last, we assume the availability of a join operator $\sqcup$ *and a widening operator* $\triangledown$*. The join operator is assumed to be sound with regard to the partial order* $\sqsubseteq$*, and the soundness of* $\triangledown$ *is specified in [10].*

2.2 Pointer Analysis

Pointer analysis can be modeled as an abstract interpretation of Imp_p.

Let *Pter* be the set of concrete states in Imp_p. Traditionally, a state $\mathsf{p} \in Pter$ is a pair of environment and heap. We write p to range over them.

$$\mathsf{p} \in Pter \triangleq (Var_p \to Ref_\perp) \times ((Ref \times Fld_p) \to Ref_\perp) \tag{4}$$

The essence of *pointer analysis* is the process of heap disambiguation, *i.e.*, the analysis partitions *Ref* into a finite set H and then summarizes the run-time pointer relations via elements h in H. The process is based on the *naming scheme.*

Definition 2. *The* naming scheme *is a mapping from concrete references to their names in* H*. The names used by the naming scheme of a pointer analysis are called* abstract references *or* abstract locations*.*

$$\rhd \in Ref \to H \tag{5}$$

We say $r \in Ref$ is abstracted by $h \in H$ if $r \rhd h$. It is required that the memory regions abstracted by different abstract references have no common concrete reference. $\forall h_1, h_2 \in H, h_1 \neq h_2 \Rightarrow \rhd^{-1}(h_1) \cap \rhd^{-1}(h_2) = \emptyset$.

This paper considers points-to analysis [15] that is widely used in heap analysis. The lattice used in the points-to abstract domain is commonly called *points-to graph*. This graph has two kinds of arcs, the unlabeled arcs from a variable to an abstract reference and the labeled arcs between abstract references that are labeled by a field. The abstract domain used in points-to analysis is a set of points-to graphs, denoted by $Pter^\sharp$.

$$Pter^\sharp \triangleq (Var_p \to \wp(H)) \times ((H \times Fld_p) \to \wp(H)) \tag{6}$$

Remark 1. Points-to analysis is based on a naming scheme that is flow independent. In other words, a given analysis pass of points-to analysis allows for a unique naming scheme, whatever the abstractions of the heap. It is worth noting that this property on the naming scheme is respected by all variants of points-to analysis (including flow-sensitive points-to analysis). In this presentation, we use a typical naming scheme to name heap elements after the program point of the statement that allocates them.

Definition 3 (Interface of traditional points-to analyzer)

$$(\mathsf{Imp}_p, \wp(Pter), [\![\cdot]\!]^\natural_p, \gamma_p, Pter^\sharp, [\![\cdot]\!]^\sharp_p)$$

The concrete domain and the abstract domain of points-to analysis are denoted by $\wp(Pter)$ and $Pter^\sharp$ respectively. They are related by a monotone concretization function $\gamma_p : Pter^\sharp \to \wp(Pter)$. *The concrete semantics is interfaced by a standard transfer function $[\![\cdot]\!]^\natural_p \in \wp(Pter) \to \wp(Pter^\sharp)$. The abstract semantics $[\![\cdot]\!]^\sharp_p \in Pter^\sharp \to Pter^\sharp$ is provided by a static numeric analyzer. This analyzer is assumed sound:*

$$[\![\cdot]\!]^\natural_p \circ \gamma_p \subseteq \gamma_p \circ [\![\cdot]\!]^\sharp_p \tag{7}$$

3 Combining Points-to and Numeric Analysis: Intuition

This section presents the intuition behind the technique of combining points-to analyses and numeric analyses. The idea is to use the names computed by the points-to analysis to create *summarized variables* that represent the numeric values stored at particular heap locations.

Example 1. Consider the Java snippet in Listing 1.1. An abstract class `Unsigned` uses unsigned numbers to represent both positive and negative values. `Unsigned` has two subclasses `Pos` and `Neg` for this purpose. It is the responsibility of clients to ensure the underlined contract, *i.e.*, the objects of type `Unsigned` must hold non-negative values. The Java source code takes an array *buf* and passes the elements to the list *elem* of type `List`. The list has a field *item* for data type `Unsigned` and a field *next* of type `List`. The compound condition structure (l. 7-14 in Listing 1.1) creates an object of class `Pos` or `Neg` according to whether n is positive or not. In both cases, *data.val* is assigned to the absolute value of n so that the assumed property of unsignedness can be preserved. From l. 15 to l. 19, the program allocates a new cell to store *data* and links it to the list created by the precedent iteration.

Below we show how we infer the following properties at the end of the program (l. 21).

- **Prop1.** Each list element of is in the range of 0 to 9:

$$\forall l \geq 0, hd.next^l.item.val \in [0, 9]$$

- **Prop2.** Each array element of *buf* is in the range of -9 to 7: $buf[*] \in [-9, 7]$.

```
1  int[] buf = {-9,7,3,-5};//h1
2  Unsigned data = null;
3  List hd = null;
4  int idx = 0;
5  while (idx < buf.length){
6   int n = buf[idx];
7   if (n > 0){
8    data = new Pos();//h2
9    data.val = n;
10  }
11  else{
12   data = new Neg();//h3
13   data.val = -n;
14  }
15  List elem = new List();//h4
16  elem.item = data;
17  elem.next = hd;
18  hd = elem;
19  idx = idx + 1;
20 }
21 return;
```

Listing 1.1. A Java snippet

```
$\delta_{h_1,[*]}$ $\doteq$ -9;
$\delta_{h_1,[*]}$ $\doteq$ 7;
$\delta_{h_1,[*]}$ $\doteq$ 3;
$\delta_{h_1,[*]}$ $\doteq$ -5;
idx = 0;
while (?){
   $n \doteq' \delta_{h_1,[*]}$;
   if (n > 0)
      $\delta_{h_2,val}$ $\doteq$ n;
      $\delta_{h_3,val}$ $\doteq$ n;
   else
      $\delta_{h_2,val}$ $\doteq$ -n;
      $\delta_{h_3,val}$ $\doteq$ -n;
   idx = idx + 1;
}
```

Listing 1.2. Semantics actions

Fig. 1. An example in Java. The program passes an array of integers to a list of `Unsigned` numbers. `Unsigned` is a superclass of `Pos` and `Neg`. It has one field `val` of integer type. The class `List` has two fields, `item` of type `Unsigned`, and `next` of type `List`.

We start with a *flow-insensitive points-to analysis*. A single points-to graph for the whole program can be obtained (Fig. 2). Semantically, the points-to graph disambiguates the heap by telling what must not alias. We derive a summarized variable $\delta_{h,val}$ for each pair of heap location h and field val. The key point is, numeric values bound to syntactically distinct summarized variables are guaranteed to be stored at different concrete heap locations. In line with the semantics of points-to graph, the analysis of the program in Listing 1.1 can be treated as an *extended* numeric analysis. This analysis is called "extended" because it not only deals with scalar variables, but also deals with summarized variables.

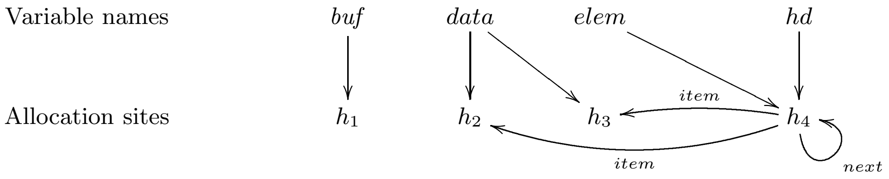


Fig. 2. A flow-insensitive points-to graph for the program in Listing 1.1

Listing 1.2 illustrates the semantics actions taken by our analysis. From l. 1 to l. 4, the summarized variable $\delta_{h_1,[*]}$ is updated with -9, 7, 3 and -5 successively. Since more than one run-time heap locations of the array buf can be associated with $\delta_{h_1,[*]}$, the semantics action is a *weak update* (denoted by $\doteq$), *i.e.*, *accumulating* values rather than *overwriting* them. The semantics action at l. 7 assigns the summarized variable $\delta_{h_1,[*]}$ to the scalar variable n. Note again that this abstract semantics should be distinguished from the abstract semantics of assignment in traditional numeric domain. This is because we should not establish a numeric relation between $\delta_{h_1,[*]}$ and n as in traditional static numeric analysis. Here we use $\doteq'$ to make a distinction. Intuitively, the assignment of $\delta_{h_1,[*]}$ to n should be abstracted as assigning the possible values of $\delta_{h_1,[*]}$ to n without coupling $\delta_{h_1,[*]}$ and n. The rest of the semantics actions in the listing should be clear now. The assignments to scalar variables at l. 5 and l. 14 are the same as in traditional numeric domains. The assignments at l. 9, 10, 12, 13 are weak update to $\delta_{h_2,val}$ and $\delta_{h_3,val}$ since both h_2 and h_3 are pointed to by the variable $data$ following the points-to graph.

By performing the extended interval analysis, we are able to infer these invariants at the end of the program: $\delta_{h_2,val} \in [0,9] \wedge \delta_{h_3,val} \in [0,9]$ and $\delta_{h_1,[*]} \in [-9,7]$, which imply **Prop1** and **Prop2** respectively.

Remark 2. The compelling part of this approach should not be the semantics actions presented so far, but the way that they can be constructed by an interplay between traditional numeric domains and points-to analysis. The advantage of this approach is that this interplay does not requires knowledge beyond the interfaces of the components in question. As demonstrated by our implementation of the analysis and its experimental results, this approach allows for direct access to many existing abstract domains including their join, widening and narrowing operators which are known difficult to implement.

4 The Language and Its Concrete Semantics

This paper focuses on how to deal with language Imp_{np}. The statements in Imp_{np} include those in Imp_n and Imp_p, and two more statements in the forms of $y_p.f_n = x_n$ and $x_n = y_p.f_n$. We write s_{np} to range over Imp_{np}.

$$s_{np} ::= s_n \mid s_p \mid y_p.f_n = x_n \mid x_n = y_p.f_n \tag{8}$$

A concrete state in Imp_{np} can be regarded as a pair of an environment and a heap

$$\begin{aligned} \mathit{State} = {} & \overbrace{(\mathit{Var}_n \to \mathbb{I}_\perp) \times (\mathit{Var}_p \to \mathit{Ref}_\perp)}^{\mathit{Env}} \\ & \times \underbrace{((\mathit{Ref} \times \mathit{Fld}_n) \to \mathbb{I}_\perp) \times ((\mathit{Ref} \times \mathit{Fld}_p) \to \mathit{Ref}_\perp)}_{\mathit{Heap}} \end{aligned} \tag{9}$$

We can turn this domain into an isomorphic shape

$$\mathit{State} \triangleq \mathit{Num}[(\mathit{Ref} \times \mathit{Fld}_n) \cup \mathit{Var}_n] \times \mathit{Pter} \tag{10}$$

where $\mathit{Num}[(\mathit{Ref} \times \mathit{Fld}_n) \cup \mathit{Var}_n]$ extends Num to $(\mathit{Ref} \times \mathit{Fld}_n) \cup \mathit{Var}_n) \to \mathbb{I}_\perp$.

Remark 3. The isomorphism consists of a crucial step. It prepares the re-use of the abstract pointer values when extending the numeric domains to cover properties about heap values.

Regarding states as (10) allows us to express the concrete semantics of Imp_{np} via those of Imp_n and Imp_p. As a shortcut, we set

$$D = \mathit{Ref} \times \mathit{Fld}_n \tag{11}$$

and use meta variable d to range over the pairs in D. In Fig. 3, we show the structural operational semantics (SOS) of Imp_{np}, denoted by $\longrightarrow^\natural$. It is expressed by $\xrightarrow{\mathit{Pter}}$ and $\xrightarrow{\mathit{Num}}$ (with $\xrightarrow{\mathit{Num}}$ in the figure extended over $D \cup \mathit{Var}_n$).

$$\frac{\langle s_n, \mathsf{n}\rangle \xrightarrow{\mathit{Num}} \mathsf{n}'}{\langle s_n, (\mathsf{n}, \mathsf{p})\rangle \longrightarrow^\natural (\mathsf{n}', \mathsf{p})} \qquad \frac{d = (\mathsf{p}(y_p), f_n) \quad \langle d = x_n, \mathsf{n}\rangle \xrightarrow{\mathit{Num}} \mathsf{n}'}{\langle y_p.f_n = x_n, (\mathsf{n}, \mathsf{p})\rangle \longrightarrow^\natural (\mathsf{n}', \mathsf{p})}$$

$$\frac{\langle s_p, \mathsf{p}\rangle \xrightarrow{\mathit{Pter}} \mathsf{p}'}{\langle s_p, (\mathsf{n}, \mathsf{p})\rangle \longrightarrow^\natural (\mathsf{n}, \mathsf{p}')} \qquad \frac{d = (\mathsf{p}(y_p), f_n) \quad \langle x_n = d, \mathsf{n}\rangle \xrightarrow{\mathit{Num}} \mathsf{n}'}{\langle x_n = y_p.f_n, (\mathsf{n}, \mathsf{p})\rangle \longrightarrow^\natural (\mathsf{n}', \mathsf{p})}$$

Fig. 3. Structural Operational semantics $\longrightarrow^\natural : \mathsf{Imp}_{np} \to \wp(\mathit{State} \times \mathit{State})$

We use the lifting of $\longrightarrow^\natural$ to the powerset $\wp(\mathit{State})$. as the collecting semantics of Imp_{np}, denoted as

$$[\![\cdot]\!]^\natural \triangleq \lambda s : \mathsf{Imp}_{np}.\mathsf{post}[\longrightarrow^\natural(s)] \tag{12}$$

5 The Abstract Domain

A state in our proposed abstract domain is a pair $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$, where $\mathsf{n}^\sharp$ is a numeric property expressed via scalar variables of Var_n and summarized variables (see below) of the set $H \times Fld_n$; the element $\mathsf{p}^\sharp$ is a lattice of $Pter^\sharp$, namely, a points-to graph in our context.

Definition 4 (Summarized variable). *A summarized variable is a pair of an abstract reference $h \in H$ and a numeric field $f_n \in Fld_n$. The set of summarized variables is denoted by Δ.*

$$\Delta \triangleq H \times Fld_n \tag{13}$$

We will use the meta-variable δ to range over the pairs in Δ, or we write δ_{h,f_n} to indicate the summarized variable corresponding to (h, f_n).

Definition 5 (The abstract domain $NumP^\sharp$). *The abstract domain $NumP^\sharp$ is defined to be*

$$NumP^\sharp \triangleq Num^\sharp[\Delta \cup Var_n] \times Pter^\sharp \tag{14}$$

Below, we specify the concretization function. It consists of an essential step before defining and proving the correctness of the abstract operators on $NumP^\sharp$.

Revisit the example in Sect. 3. We have obtained the state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ at the end of the program, with

$$\mathsf{n}^\sharp = \{\delta_{h_2,val} \in [0, 9], \delta_{h_3,val} \in [0, 9], \delta_{h_1,[*]} \in [-9, 7]\} \tag{15}$$

and $\mathsf{p}^\sharp$ is the points-to graph specified in Fig. 2. A concrete state $(\mathsf{n}, \mathsf{p}) \in State$ is in the concretization of $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ if for any reference r,

– we have $\mathsf{n}(r, val) \in [0, 9]$ as long as r is abstracted by h_2, *i.e.*, $r \rhd h_2$, and
– we have $\mathsf{n}(r, val) \in [0, 9]$ as long as r is abstracted by h_3, *i.e.*, $r \rhd h_3$, and
– we have $\mathsf{n}(r, [*]) \in [-9, 7]$ as long as r is abstracted by h_1, *i.e.*, $r \rhd h_1$

and p has to be a concrete state abstracted by $\mathsf{p}^\sharp$, *i.e.*, $\mathsf{p} \in \gamma_p(\mathsf{p}^\sharp)$. By abuse of language, we have treated the array index $[*]$ above as an aggregate numeric field. In other words, we say (n, p) is in the concretization of $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ if n is in the concretization of all $\mathsf{n}^{\sharp\prime}$ that is the numeric property $\mathsf{n}^\sharp$ with each of its summarized variables δ substituted by some d of $Ref \times Fld_n$ (namely D) that satisfies $\rhd(d) = \delta$ (with $\rhd$ extended by taking care of numeric fields).

Definition 6 (Instantiation). *Let $\rhd$ be naming scheme that is extended from $Ref \to H$ to $Ref \times Fld_n \to H \times Fld_n$. We define the space of instantiation as a set of mappings from Δ to D.*

$$\mathtt{Ins}_\rhd \triangleq \{\sigma : \Delta \to D \mid \sigma(h, f_n) = (r, g_n) \Rightarrow h = \rhd(r) \wedge f_n = g_n\} \tag{16}$$

Definition 7. *The concretization function γ_{np} of $NumP^\sharp \to \wp(State)$ is defined as*

$$\gamma_{np}(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \{(\mathsf{n}, \mathsf{p}) \mid \mathsf{p} \in \gamma_p(\mathsf{p}^\sharp) \wedge \forall \sigma \in \mathtt{Ins}_\triangleright : \mathsf{n} \in \gamma_n \circ [\sigma](\mathsf{n}^\sharp)\} \tag{17}$$

where we denote by $[\sigma]$ the capture-avoiding substitution operator that replaces all the free occurrences of δ in $\mathsf{n}^\sharp \in Num^\sharp[\Delta \cup Var_n]$ with $\sigma(\delta)$.

Example 2. Consider the following program:

```
List hd = null, tmp;
int i;
for (i = -17;i < 42;i++){
  List tmp = new List(); // allocation site h
  tmp.val = i;
  tmp.next = hd;
  hd = tmp;
}
```

A list of integers ranging from -17 to 41 is stored iteratively on the heap. At each iteration, a memory cell bound to variable *tmp* is allocated. The cell consists of a numeric field *val* and a reference field *next*. The head of the list is always pointed to by the variable *hd*.

The abstract memory state computed at the end of program is given by

$$(\mathsf{n}^\sharp, \mathsf{p}^\sharp) = \left(\{\delta_{h,val} \in [-17, 41], i = 42\} \qquad \begin{tikzcd} \text{tmp} \arrow[r] & \text{h} \arrow[loop right, "next"] \\ \text{hd} \arrow[ur] & \end{tikzcd} \right) \tag{18}$$

5.1 Transfer Functions

Let $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ be a state of $NumP^\sharp$. We are concerned with how it should be updated by statements of Imp_{np}.

Transfer Function for s_n. It is sound to assume that assignments or assertions of numeric variables have no effect on the heap. If s_n is an assignment in Imp_n, it can be treated in the same way as in traditional numeric analysis using its abstract transfer function $[\![\cdot]\!]_n^\sharp$ (as specified in Sect. 2.1).

The transfer function for updating $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with s_n can be defined as:

$$[\![s_n]\!]^\sharp (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq [\![s_n]\!]_n^\sharp\, \mathsf{n}^\sharp, \mathsf{p}^\sharp \tag{19}$$

If s_n is an assertion in Imp_n, $\mathsf{p}^\sharp$ may be refined. For example, consider the *compound statement*[1] `if (a > 0) p = q` where p and q are reference variables and a is a numeric variable. Although it should be possible to perform a dead-code elimination using inferred numeric relations, similar to Pioli's conditional constant propagation [25], we still use the Eq. (19) for the ease of implementation.

[1] This term is used here to be distinguished from basic statements as s_n, s_p or s_{np}. Note that s_n is the assertion, not the whole if-statement.

Transfer Function for s_p. It is also sound to assume that s_p has no effect upon $\mathsf{n}^\sharp$. Yet the reasoning is different from the above case. For example, if $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ is the state shown on Eq. (18), how can we tell whether an assignment of pointers operation modifies $\mathsf{n}^\sharp$ or not? Recall that the intended semantics of $\delta_{h,val} \rightarrow [17, 41]$ is that every value stored in each (r, val) satisfying $\triangleright(r) = h$ must be in the range of $[-17, 41]$. That is to say, $\mathsf{n}^\sharp$ represents a fact about the *numeric content* stored in the corresponding concrete references. Since a pointer assignment can by no means modify any numeric values stored in the heap, the algorithm to update $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with s_p can be written as:

$$[\![s_p]\!]^\sharp (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \mathsf{n}^\sharp, [\![s_p]\!]^\sharp_p \, \mathsf{p}^\sharp \tag{20}$$

Transfer Function of $y_p.f_n = x_n$. Consider an assignment $y_p.f_n = x_n$ with y_p pointing to $h \in H$. We regard $y_p.f_n = x_n$ as an weak update to summarized variable δ_{h,f_n}, That is, the field f_n of *one of* the concrete objects represented by h is to be updated with the value of x_n, while the other concrete objects represented by h remain unchanged. This effect can be approximated by $\lambda \mathsf{n}^\sharp . \mathsf{n}^\sharp \sqcup [\![\delta_{h,f_n} = x_n]\!]^\sharp_n (\mathsf{n}^\sharp)$. Below, we write

$$\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta \tag{21}$$

if δ is associated with (h, val) and y_p points to h. The transfer function of $y_p.f_n = x_n$ can be modeled by joining the effects of weak update of all δ by x_n such that $\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta$.

$$[\![y_p.f_n = x_n]\!]^\sharp (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \left(\left(\bigsqcup_{\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta} \mathsf{n}^\sharp \sqcup [\![\delta = x_n]\!]^\sharp_n (\mathsf{n}^\sharp) \right), \mathsf{p}^\sharp \right) \tag{22}$$

Note that it is not necessary to compute transfer functions for assertions involving field expressions for they are transformed beforehand by our front-end SOOT to assertions in Imp_n or in Imp_p. For instance, a source code `if (x.f > 0) ...`, is transformed to `a = x.f; if (a > 0) ...` before our analysis.

Transfer Function of $x_n = y_p.f_n$. Consider the snippet

```
a = x.f; b = y.f; if (a < b) {...}
```

Assume that $\mathsf{p}^\sharp \vdash x.f \Downarrow \delta$ and $\mathsf{p}^\sharp \vdash y.f \Downarrow \delta$. It is tempting, but wrong, to abstract the semantics of $a = x.f$ (resp. $b = y.f$) as $[\![a = \delta]\!]^\sharp_n$ (resp. $[\![b = \delta]\!]^\sharp_n$) following which the analysis would incorrectly argue that the `if` branch can never be reached.

This issue was carefully studied and solved by Gopan *et al.* [18]. The authors showed that it would be wrong to correlate a summarized dimension δ to a non-summarized dimension x_n even if the former is assigned to the later; they argued that the correct way to assign a summarized dimension δ to a non-summarized dimension x_n takes three steps: first, copy the summarized dimension δ to a fresh δ', and then relate x_n with δ' using traditional abstract semantics for assignment. Finally, the newly introduced dimension δ' has to be removed. Intuitively, the resulting abstract value keeps the possible (abstract) values of δ without being correlated with it. Gopan *et al.* have introduced four non-standard operators, in particular, "drop" that removes dimensions, and "expand" that copies dimensions. We use

$$[\![x_n = y_p.f_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \bigsqcup_{\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta} G(x_n, \delta)\ \mathsf{n}^\sharp, \quad \mathsf{p}^\sharp \tag{23}$$

where Gopan's operator $G(x_n, \delta)$ is the composition of the three steps described above:

$$G(x_n, \delta) \triangleq \lambda \mathsf{n}^\sharp.\, \mathrm{drop}^\sharp_{\delta'} \circ [\![x_n = \delta']\!]^\sharp_n \circ \mathrm{expand}^\sharp_{\delta,\delta'}\ \mathsf{n}^\sharp \tag{24}$$

Above, we assume dimension δ' does not belong to the dimensions of $\mathsf{n}^\sharp$ in question.

Example 3. Let $\mathsf{n}^\sharp = \delta \rightarrow [0, 1]$. Even if we use a relational domain like polyhedral analysis, only $G(x, \delta)\mathsf{n}^\sharp = x \rightarrow [0, 1], \delta \rightarrow [0, 1]$ can be obtained, while traditional numeric domains would establish a relationship between x and δ.

Theorem 1 (Soundness). *The transfer functions* $[\![\cdot]\!]^\sharp : \mathsf{Imp}_{np} \rightarrow (\mathit{NumP}^\sharp \rightarrow \mathit{NumP}^\sharp)$*, defined in* (19)*,* (20)*,* (22) *and* (23)*, are sound with respect to* $[\![\cdot]\!]^\natural$*: for any statement* s *of* Imp_{np} *and abstract state* $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ *of* $\mathit{NumP}^\sharp$*,* $[\![s]\!]^\natural \circ \gamma_{np}(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \subseteq \gamma_{np} \circ [\![s]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$*.*

We give a proof sketch for the case of $[\![x_p.f_n = y_n]\!]^\sharp$. It is important to note that the soundness of the theorem is based on the soundness hypotheses of $[\![\cdot]\!]^\sharp_n$ and $[\![\cdot]\!]^\sharp_p$. The combined analysis is sound as long as its component analyses are.

Proof. For all $\mathsf{n}^\sharp \in \mathit{Num}^\sharp[\Delta \cup \mathit{Var}_n]$ and $\mathsf{p}^\sharp \in \mathit{Pter}^\sharp$, we prove

$$[\![x_p.f_n = y_n]\!]^\natural\,(\gamma_{np}(\mathsf{n}^\sharp, \mathsf{p}^\sharp))\ \ \dot{\subseteq}\ \ \gamma_{np}([\![x_p.f_n = y_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp)) \tag{25}$$

By the definitions of $[\![x_p.f_n = y_n]\!]^\natural$ and $[\![x_p.f_n = y_n]\!]^\sharp$ and the monotony of γ_δ, it is sufficient to show for any d such that $\gamma_p(\mathsf{p}^\sharp) \vdash x_p.f_n \Downarrow d$, we have

$$[\![d = y_n]\!]^\natural_n \circ \gamma_\delta(\mathsf{n}^\sharp) \subseteq \gamma_\delta(\mathsf{n}^\sharp \sqcup [\![\delta = y_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)) \tag{26}$$

where we note $\delta = \rhd(d)$.

By the definition of γ_δ, it is then sufficient to prove a stronger condition:

$$\forall \sigma \in \mathtt{Ins}_\rhd : [\![d = y_n]\!]^\natural_n \circ \gamma_n \circ [\sigma](\mathsf{n}^\sharp) \subseteq \gamma_n \circ [\sigma](\mathsf{n}^\sharp) \cup \gamma_n \circ [\sigma]([\![\delta = y_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)) \tag{27}$$

Given an instantiation σ (as defined in Eq. (6)), we make two cases to conclude:

- **Case I:** σ does not map δ to d. By consequence d does not appear in $[\sigma](\mathsf{n}^\sharp)$ and $\llbracket d = y_n \rrbracket_n^\sharp \circ \gamma_n \circ [\sigma](\mathsf{n}^\sharp) = \gamma_n \circ [\sigma](\mathsf{n}^\sharp)$. This concludes this case.
- **Case II:** σ maps δ to d. We can then simplify the right part of (27) because $[\sigma](\llbracket \delta = y_n \rrbracket_n^\sharp (\mathsf{n}^\sharp)) = (\llbracket d = y_n \rrbracket_n^\sharp \circ [\sigma](\mathsf{n}^\sharp))$. We then conclude this last case using the soundness of $\llbracket d = y_n \rrbracket_n^\sharp$.

5.2 Join and Widening

The join of two facts is defined as the set of all facts that are implied independently by both. Thanks to our hypothesis of flow independent naming scheme (in Sect. 2.2), the join and widening of $NumP^\sharp$ are easy to define: we just have to compute the join (or widening) component wise. Then, if a concrete state (n, p) is in $\gamma_{np}(\mathsf{n}_1^\sharp, \mathsf{p}_1^\sharp)$ or $\gamma_{np}(\mathsf{n}_2^\sharp, \mathsf{p}_2^\sharp)$, it is also in the concretization of $(\mathsf{n}_1^\sharp \sqcup \mathsf{n}_2^\sharp, \mathsf{p}_1^\sharp \cup \mathsf{p}_2^\sharp)$. Thus the join of $(\mathsf{n}_1^\sharp, \mathsf{p}_1^\sharp)$ and $(\mathsf{n}_2^\sharp, \mathsf{p}_2^\sharp)$ is the join of $\mathsf{n}_1^\sharp$ and $\mathsf{n}_2^\sharp$, paired with the join of $\mathsf{p}_1^\sharp$ and $\mathsf{p}_2^\sharp$ (Sect. 2). The case for widening is similar.

$$(\mathsf{n}_1^\sharp, \mathsf{p}_1^\sharp) \sqcup^\sharp (\mathsf{n}_2^\sharp, \mathsf{p}_2^\sharp) = (\mathsf{n}_1^\sharp \sqcup \mathsf{n}_2^\sharp, \mathsf{p}_1^\sharp \cup \mathsf{p}_2^\sharp) \tag{28}$$

$$(\mathsf{n}_1^\sharp, \mathsf{p}_1^\sharp) \triangledown^\sharp (\mathsf{n}_2^\sharp, \mathsf{p}_2^\sharp) = (\mathsf{n}_1^\sharp \triangledown \mathsf{n}_2^\sharp, \mathsf{p}_1^\sharp \cup \mathsf{p}_2^\sharp) \tag{29}$$

5.3 Constraint System with a Flow-Insensitive Points-to Analysis

In our implementation, we use a flow-insensitive points-to analysis as a pre-analysis step. It is worth nothing that using flow-insensitive variant does not cause any soundness issue. This is because the soundness of our analysis is based on the soundness of its component numeric domains and pointer analysis; taking the flow-insensitive points-to graph during all propagation can be modeled as an analysis that is initialized with a set that is larger than the least fix point of a flow-sensitive analysis, and propagates in the style of `skip`, which satisfies the soundness requirement for the pointer analysis component.

Let $F^\sharp(s) \triangleq \lambda \mathsf{n}^\sharp.\mathsf{fst} \circ \llbracket s \rrbracket^\sharp (\mathsf{n}^\sharp, \mathsf{p}_{fi}^\sharp)$, where $\mathsf{p}_{fi}^\sharp$ is the flow-insensitive points-to graph, and fst is the operator that extracts the first element from a pair of components. We use the following constraint system that operates on numeric lattice $\mathsf{n}^\sharp$ only (rather than on $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ pair):

$$\overline{\mathsf{n}^\sharp}[l] \sqsupseteq F^\sharp(s)(\overline{\mathsf{n}^\sharp}[l']) \tag{30}$$

where we write $\overline{\mathsf{n}^\sharp}[l]$ (resp. $\overline{\mathsf{n}^\sharp}[l']$) for the numeric component of $NumP^\sharp$ at control point l (resp. l'), l' being the control point of statement s, and (l', l) is an arc of the program control flow.

Example 4. Consider the Java snippet in Fig. 4. From l. 4 to l. 10 is the same as in the example program of Sect. 4. Since we do not propagate the points-to graph here, the state at l. 10 is the numeric lattice $\mathsf{n}_0^\sharp$:

$$\mathsf{n}_0^\sharp = \{\delta_{h,val} \to [-17, 41], i \to 42, max \to \top, n \to \top\} \tag{31}$$

where three scalar variables i, max and n as well as a summarized variable $\delta_{h,val}$ are involved. Note that the flow-insensitive points-to graph

$$\begin{tikzcd}[row sep=tiny] & \text{tmp} & \\ \mathsf{p}^\sharp_{fi} = & \text{hd} \arrow[r] & \text{h} \arrow[loop below, "next"] \\ & \text{cur} & \end{tikzcd} \tag{32}$$

is used in the process of propagation of states but the points-to graph itself will keep unchanged (as formalized in (30)). From l. 14 to l. 21, the program finds the maximal value from the list. This value is then stored in the variable max. In case there is no positive value or the list is empty, max takes its initial value 0. We will show that at the end of the program, (l. 10):

– the scalar value max has to be in the range of $[0, 41]$

The propagation of states from lattice $\mathsf{n}^\sharp_0$ is shown in Fig. 5.

```
1  //create a list of integers
2  List hd = null, cur, tmp;
3  int i, n, max;
4  for (i = -17;i < 42;i++){
5    List tmp = new List();
   // h
6    tmp.val = i;
7    tmp.next = hd;
8    hd = tmp;
9  }
10
11 //find the maximum
12 cur = hd;
13 max = 0;
14 while (cur != null){
15   n = cur.val;
16   if (max < n){
17     max = n;
18   }
19   cur = cur.next;
20 }
21
```

Fig. 4. An example in Java. The class List has *val* and *next* as fields.

6 Experiments

We have implemented a prototype for the abstract domain $NumP^\sharp$. The implementation is called NumP. This section presents the prototype and our experimental results.

The input Java program is passed to SOOT. It computes the points-to graph and transforms the program to Jimple IR [30]. The analysis combines the abstract domains from PPL and the points-to analysis in SOOT. It infers numeric properties for each program point of the IR.

The analyzer NumP combines PPL and SOOT in a modular way. We first implement the traditional static numeric analyzer for Java. The implementation is denoted by Num, which is implemented by wrapping abstract domains in PPL. Num either skips unrecognized statements or conservatively approximates them using the `unconstraint` operator in PPL. The re-used components in SOOT include notably the flow-insensitive points-to analysis (from its SPARK toolkit [20]). This analyzer is denoted by Pter subsequently.

```
$\delta \to [-17,41],\ i \to 42,\ max \to \top,\ n \to \top$
cur = hd;
$\delta \to [-17,41],\ i \to 42,\ max \to \top,\ n \to \top$
max = 0;
 $\delta \to [-17,41],\ i \to 42,\ max \to 0,\ n \to \top$
while (hd != null){
$\delta \to [-17,41],\ i \to 42,\ max \to 0,\ n \to \top$
$\delta \to [-17,41],\ i \to 42,\ max \to [0,41],\ n \to \top$
    n = hd.val;
    $\delta \to [-17,41],\ i \to 42,\ max \to 0,\ n \to [-17,41]$
    $\delta \to [-17,41],\ i \to 42,\ max \to [0,41],\ n \to [-17,41]$
    if (max < n){
        $\delta \to [-17,41],\ i \to 42,\ max \to 0,\ n \to [1,41]$
        $\delta \to [-17,41],\ i \to 42,\ max \to [0,41],\ n \to [1,41]$
        max = n;
        $\delta \to [-17,41],\ i \to 42,\ max \to [1,41],\ n \to [1,41]$
    }
    $\delta \to [-17,41],\ i \to 42,\ max \to [0,41],\ n \to [-17,41]$
    hd = hd.next;
    $\delta \to [-17,41],\ i \to 42,\ max \to [0,41],\ n \to [-17,41]$
}
$\delta \to [-17,41],\ i \to 42,\ max \to [0,41],\ n \to [0,41]$
```

Fig. 5. The propagation of states from l. 14 to l. 21 of the program in Fig. 4. The fixpoint is reached in two steps.

To demonstrate the effectiveness of our technique, we evaluate the analyzer on Dacapo-2006-MR2 [4] benchmark suite. The experiments were performed on a 3.06 GHz Intel Core 2 Duo with 4 GB of DDR3 RAM laptop with JDK 1.6. We tested *all* 11 benchmarks in Dacapo.

Experimental results are shown in Tab. 1 using the interval domain `Int64_Box` from PPL and the flow-insensitive points-to analysis from SOOT. The characteristics of the benchmarks are presented by the number of analyzed Jimple statements (col. 2, STATEMENT) and the number of write access statements in the form of $y_p.f_n = x_n$ or $y_p.f_n = k$ with k being a constant (col. 3, WA).

We measure PRCS (col. 4) for the number of the write access statements after which the obtained invariants are strictly more precise than Num. Q_PRCS (col. 5) is the ratio of PRCS and WA

$$\text{Q\_PRCS} \triangleq \text{PRCS}/\text{WA} \tag{33}$$

We record Q_PRCS as the metric for precision enhancement of the analyzer.

The execution time is measured for Num, Pter and NumP (col. 8, 9 and 10). The parameters T_Num and T_Pter are the times spent by Num and Pter when they analyze individually. The parameter T_NumP records the time spent our combined analysis instantiated with the interval and flow-insensitive points-to analysis.

Table 1. Evaluation of NumP on the benchmark suite Dacapo-2006-MR2

Benchmark Characteristics			Precision		Time			
BENCHMARK	STATEMENT	WA	PRCS	Q_PRCS	T_NUM	T_PTER	T_NUMP	Q_T
antlr	26776	766	174	23%	00m29s	00m53s	01m36s	117%
bloat	64328	2472	943	38%	01m35s	01m02s	16m33s	632%
chart	132627	10244	3690	36%	04m17s	13m20s	83m21s	473%
eclipse	56772	820	116	14%	00m46s	00m54s	01m52s	112%
fop	198541	23482	6166	26%	03m25s	05m11s	275m28s	3203%
jython	88302	2583	1356	52%	00m57s	01m04s	05m38s	279%
hsqldb	6286	352	10	3%	00m19s	00m49s	01m16s	112%
luindex	22192	1206	250	21%	00m33s	00m54s	01m31s	105%
lusearch	26711	1503	418	28%	00m38s	00m56s	01m35s	101%
pmd	80640	3675	1316	36%	00m50s	00m55s	04m25s	252%
xalan	5197	341	3	1%	00m16s	00m49s	01m12s	111%
Mean	64397	4313	1313	25%	01m17s	02m26s	35m52s	500%

The last column Q_T evaluates the time overhead of our analyzer. It is computed as the ratio of the time spent by our analysis to the total time spent by its component analyses.

$$\text{Q\_T} \triangleq \text{T\_NumP}/(\text{T\_Num} + \text{T\_Pter}) \tag{34}$$

The size of the analyzed Jimple statements ranges from 5,197 (`xalan`) to 198,541 (`fop`). The average precision metric is given in the last row of Tab. 1. The mean Q_PRCS (25%) shows a clear precision enhancement of our approach over numeric analysis only. The time overheads Q_T are generally acceptable.

In summary, we have designed an analysis in a modular way. It can be scaled to real-life programs; analyzing programs of hundreds of thousands of lines within hours can be a reasonable time budget for many applications. The precision enhancement is validated in practice.

7 Related Work

Static analysis of numeric properties has been extensively studied, especially in the framework of abstract interpretation [11]. While a large number of articles covers issues related to numeric abstractions, program analyses where both pointers and numeric values are taken into account are comparatively few.

The back-end of CodePeer[2] takes a flow-insensitive may-aliasing analysis to distinguish heap objects and to transform the analyzed programs to their SSA forms using the global value numbering technique. The value propagation of CodePeer infers the value ranges of subtraction of variables, in other words, properties of the zone abstract domain. CodePeer goes further by taking care of

[2] `http://www.adacore.com/codepeer`

inductive loop variables and the disjunctive numeric constraints, so that properties such as $b > 0 \Rightarrow a = 2 * b$ can be inferred where a or b is an inductive scalar variable. Compared with our approach, however, CodePeer uses a single zone abstract domain and do not offer the flexibility to easily plug in other abstract domains of different precision/cost tradeoffs such as the more efficient interval abstract domain or the more precise polyhedral domain. In our approach, even the capability of expressing disjunctive facts in CodePeer can be easily implemented by instantiating our numeric domain component as the powerset construction domains [2].

Efforts have been made to parametrize numeric domains with a dedicate pointer analysis. Fähndrich and Logozzo's Clousot analyzer [16] uses a value numbering algorithm to compute an under-approximation of must-alias. An optimistic assumption is then made so that Clousot regards two access paths not aliased if they do not have the same value numbering.[3] The ASTREE static analyzer [3] relies on a type based pointer analysis to deal with numeric properties of heap objects. The abstraction can be used with pointer arithmetic, union types and records of stack variables in C programs that do not have dynamic memory allocation or recursive structure. This category of static analyzers, as well as ours, can be regarded as applications of the theory of abstract domain combination which has been thoroughly studied and applied in many other contexts [28,12,8].

A more sophisticated heap abstraction is *shape analysis* [26]. The TVLA [19] framework based on shape analysis uses *canonical abstraction* to create bounded-size representations of memory states. The analyses of this family are precise and expressive. TVLA users are demanded to specify the concrete heap using first-order predicates with transitive closure, or user-defined *instrumentation predicates* like `IsNotNull`. Then TVLA automatically derives an abstract semantics based on the users' specification. The numeric abstraction of Gopan *et al.* [18] allows the integration of TVLA with existing numeric domains. The static verifier DESKCHECK [21] combines TVLA and numeric domains. It is sufficiently precise and expressive to check quantified invariants over both heap objects and numeric values. Besides the burden for users to specify the program (a problem that XISA [7,6] attempts to remedy), the major issue of the shape-analysis-based approaches lies in their scalability. In contrast, our experiments show our capability to run over large programs.

Pioli and Hind [25] show the mutual dependence of *conditional constant analysis* and pointer analysis. The combination is specifically designed for the conditional constant analysis and is not generalized to standard numeric domains. In particular, this approach does not directly cooperate with standard numeric domains because their method relies on the particular feature of conditional constant analysis that is able to partially eliminate infeasible branches.

In a somewhat different strand of work, numeric domains have been used to enhance pointer analysis. Deutsch [14] uses a parametrized numeric domain to

[3] This assumption is said optimistic because it is possible two access paths alias at run-time but are considered never aliased by Clousot.

improve the accuracy of alias analysis in the presence of recursive pointer data structures. The key idea is to quantify the symbolic field references with integer coefficients denoting positions in data structures. This analysis is able to express properties for cyclic structures such as "for any k, the k-th element of list l of length len, is aliased to its $(k + len)$-th element". Venet [31] develops the structure called the *abstract fiber bundle* to formalize the idea of embedding an abstract numeric lattice within a symbolic structure. The structure enables the using of the large number of existing numeric abstractions to encode a broad spectrum of symbolic properties.

8 Conclusion

The primary objective of this work has been the automatic discovery of numeric invariants in Java-like programs, which are generally pointer-aware. We have proposed a methodology for combining numeric analyses and points-to analysis, developed using an approach based on concepts from abstract interpretation. In particular, we have shown how the abstract domain used in points-to analysis can be used to lift a numeric domain to encompass values stored in the heap. The new abstract domain and the accompanying transfer functions have been specified formally and their correctness proved. Moreover, the modular way in which the abstract domains are combined via some well-defined interfaces is reflected in the modular construction of a prototype implementation of the analysis framework. This modularity has enabled us to experiment with different choices for the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators. Concretely, the derived abstract semantics allows us to combine existing numeric domains (interval domains, octagon etc.) with existing points-to analyses. The modular analyzer is able to combine advanced libraries as PPL and SPARK and it shows a clear precision enhancement with low time overhead.

Acknowledgments. The author wishes to express his gratitude to Thomas Jensen, Laurent Mauborgne and David Pichardie for their thoughtful feedback.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Technical Report 457, Dipartimento di Matematica, Università di Parma, Italy (2006)
2. Bagnara, R.: A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. Sci. Comput. Program. 30(1-2), 119–155 (1998)
3. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis by abstract interpretation of embedded critical software. ACM SIGSOFT Software Engineering Notes 36(1), 1–8 (2011)

4. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 169–190. ACM Press, New York (2006)
5. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6), 26 (2011)
6. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: POPL, pp. 247–260 (2008)
7. 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)
8. Cortesi, A., Le Charlier, B., Van Hentenryck, P.: Combinations of abstract domains for logic programming: open product and generic pattern construction. Sci. Comput. Program. 38(1-3), 27–71 (2000)
9. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, pp. 106–130, Dunod, Paris (1976)
10. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
11. 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)
12. 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)
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
14. Deutsch, A.: A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In: ICCL, pp. 2–13 (1992)
15. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI, pp. 242–256 (1994)
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. Fu, Z.: Static Analysis of Numerical Properties in the Presence of Pointers. PhD thesis, Université de Rennes 1 – INRIA, France (2013)
18. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 512–529. Springer, Heidelberg (2004)
19. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)
20. Lhoták, O., Hendren, L.: Scaling java points-to analysis using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
21. 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)

22. Miné, A.: Weakly Relational Numerical Abstract Domains. PhD thesis, École Polytechnique, Palaiseau, France (December 2004)
23. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: LCTES, pp. 54–63 (2006)
24. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
25. Pioli, A., Hind, M.: Combining interprocedural pointer analysis and conditional constant propagation. Technical report, IBM T. J. Watson Research Center (1999)
26. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, pp. 105–118. ACM, New York (1999)
27. Simon, A.: Value-Range Analysis of C Programs. Springer (August 2008)
28. Toubhans, A., Chang, B.-Y.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 375–395. Springer, Heidelberg (2013)
29. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1999, p. 13. IBM Press (1999)
30. Vallee-Rai, R., Hendren, L.J.: Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University (July 1998)
31. Venet, A.: Towards the integration of symbolic and numerical static analysis. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 227–236. Springer, Heidelberg (2008)
32. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

Generic Combination of Heap and Value Analyses in Abstract Interpretation

Pietro Ferrara[1,2]

[1] IBM Thomas J. Watson Research Center, USA
[2] ETH Zurich, Switzerland
pietroferrara@us.ibm.com

Abstract. Abstract interpretation has been widely applied to approximate data structures and (usually numerical) value information. One needs to combine them to effectively apply static analysis to real software. Nevertheless, they have been studied mainly as orthogonal problems so far. In this context, we introduce a generic framework that, given a heap and a value analysis, combines them, and we formally prove its soundness. The heap analysis approximates concrete locations with heap identifiers, that can be materialized or merged. Meanwhile, the value analysis tracks information both on variable and heap identifiers, taking into account when heap identifiers are merged or materialized. We show how existing pointer and shape analyses, as well as numerical domains, can be plugged in our framework. As far as we know, this is the first sound generic automatic framework combining heap and value analyses that allows to freely manage heap identifiers.

1 Introduction

Two major fields of static program analysis have been heap and (usually numerical) value abstractions. Venet states that "If one wants to use static analysis to support or achieve verification of *real* programs, we believe that symbolic (i.e., heap) and numerical static analysis must be tightly integrated"[33]. Nevertheless, "symbolic and numerical static analysis are commonly regarded as entirely orthogonal problems".

Object-oriented programming languages are currently mainstream in software development, and many analyzers targeting these languages have been developed. Two main lines appeared in this context: (i) analyzers focused on value information that preprocess the program applying a specific heap analysis, and replace heap accesses with symbolic variables (e.g., Clousot [23]), and (ii) heap abstractions (e.g., TVLA [22]) that do not track value information, or that have to be manually extended (e.g., with specific predicates) to track a particular type of value information [24,25]. As far as we know, existing analyzers that combine heap and value analyses are not both generic (that is, they are specific on a particular heap and/or value analysis) and automatic (that is, they require to provide some annotation, like instrumentation predicates).

Motivating Example. Consider the motivating example in Figure 1. Class `ListInt` represents a list of integers, with an integer field `f` (containing the value of an element) and a `ListInt next` field (pointing to the next element of the list, or to `null` if we are at the end). Method `absSum(l)` computes the sum of the absolute values of the elements in the list. Imagine that two clients call this method. `client1` passes the list [1; 2][1] to `absSum`, where the two elements are allocated at different program points (`p1` and `p2`). Instead, `client2` calls `absSum` with a list of `n` positive elements, where `n` is an input of the program. There are various properties and invariants we would like to prove and infer on such program. First of all, we would like to prove that we do not have any `NullPointerException` (property `P1`). In addition, we could discover that the value returned by `sumAbs` is positive (`P2`), or that it is greater or equal than all the elements in the list pointed by `l` (`P3`). These properties require to combine different heap and value analyses. `P1` does not require any particular numerical analysis, and for both the clients a simple and efficient heap analysis based on the allocation sites [29] would be precise enough. Instead, `P2` requires at least a numerical domain that tracks the sign of numerical variables, while `P3` requires a relational domain like Octagons [28]. In addition, for `client1` the allocation site-based heap abstraction would be precise enough both for `P2` and `P3`. Instead, on `client2` this abstraction would approximate all the nodes of the list with a unique summary node, and it would not be able to discover that the value added to `sum` is positive, since it cannot track precise information on the Boolean condition of the `if` statement. Therefore, we need a more precise heap abstraction that materializes the node pointed by `it` (e.g., shape analysis [30]).

```
int absSum(ListInt l) {
    int sum = 0;
    ListInt it = l;
    while(it != null) {
        if(it.f < 0) sum = sum - it.f;
        else sum = sum + it.f;
        it=it.next;
    }
    return sum;
}
```

Fig. 1. The motivating example

Contribution. The contribution of this work is the formalization of a sound generic analysis that allows to combine various heap and value abstractions automatically. The heap analysis approximates concrete locations through *heap identifiers*, while the value analysis tracks information on these identifiers. In addition, our framework allows the heap analysis to *freely* manage heap identifiers, and in particular to merge and materialize them. These modifications are represented by *substitutions*, and they are propagated to the value analysis. For the most part, our approach relies on standard components of abstract interpretation-based sound static analyses, and we formally define and prove the soundness of their combination. In addition, we show how to instantiate our framework with a pointer and a shape analyses, as well as with numerical

[1] [1; 2] is a shortcut to denote a list of two elements, with value 1 stored in the field f of the first element list, and 2 in the second one.

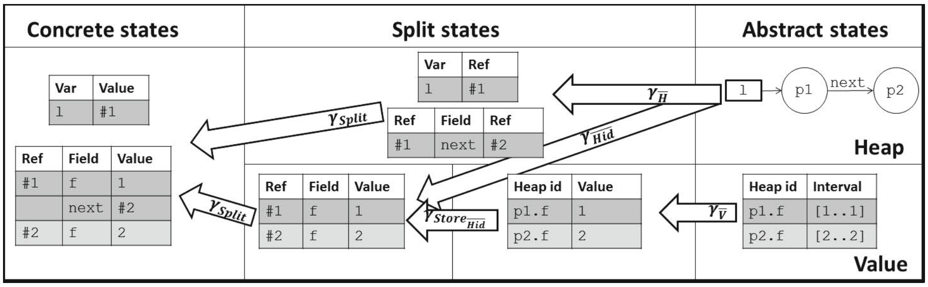


Fig. 2. The architecture of the domains in our approach

domains. This proves that our framework is expressive enough to be applied to the most common heap and value analyses.

1.1 Overview of the Framework

Domains. Figure 2 depicts the overall structure of our approach. On the left, we have standard object-oriented states composed by an environment and a store. On the right, we have our target abstract domain composed by a heap and a value abstract state. Here we represent the state of `client1` when calling method `absSum` in our motivating example. We adopt an allocation site-based heap abstraction [1] and the Interval domain [9]. Therefore, the heap analysis abstracts the list with two abstract nodes named `p1` and `p2`, while the value analysis tracks that field `f` of `p1` is [1..1], and field `f` of `p2` is [2..2].

The heap analysis concretizes to a set of environments and stores representing information only about references. This is represented in Figure 2 in the upper central box of *split* states, and it is obtained through the heap concretization $\gamma_{\overline{\mathsf{H}}}$. Similarly, the value analysis concretizes to environments and stores representing information only about values through $\gamma_{\overline{\mathsf{V}}}$. The value analysis contains information about heap identifiers `p1.f` and `p2.f`, and it needs the concretization of heap identifiers $\gamma_{\overline{\mathsf{HId}}}$ provided by $\gamma_{\overline{\mathsf{H}}}$ to produce concrete states. For instance, in Figure 2 we assume that one possible $\gamma_{\overline{\mathsf{HId}}}$ concretizes `p1.f` and `p2.f` to $(\#1, \texttt{f})$ and $(\#2, \texttt{f})$, respectively.

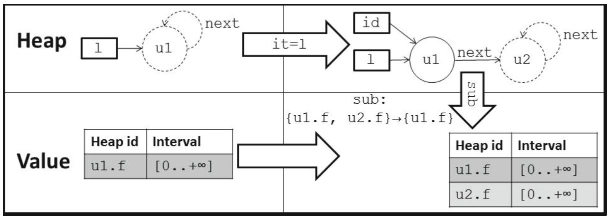


Fig. 3. The semantics' architecture of our approach

Semantics. The heap semantics may need to materialize or merge heap identifiers. The information about how the heap identifiers are modified is communicated through *substitutions*. A substitution is a function that tells the value analysis from which heap identifiers of the pre-state the identifiers of the post-state come. In this way, the value analysis can preserve the soundness of the information tracked on heap identifiers modified by the heap analysis.

For instance, consider the semantics step depicted in Figure 3. Suppose to analyze our motivating example combining a simple shape analysis [30] with the Interval domain. We analyze method `absSum` when it is called by `client2`. The abstract state on the left of Figure 3 is produced before computing the abstract semantics of line `3` in Figure 1[2]. Supposing that the node pointed by `it` has to be always definite, when we assign `l` to `it`, a definite node is materialized from `u1`. The value analysis was already tracking information about `u1.f` before this step, and it has to propagate this information to the materialized identifier `u2.f`. The heap analysis communicates a substitution `sub` to the value analysis, telling that `u1.f` and `u2.f` in the post-state derive from `u1.f` in the pre-state. Then the value analysis is updated reflecting the semantics of this substitution, that is, assigning the value tracked on `u1.f` in the pre-state to `u1.f` and `u2.f` in the post-state.

This paper formalizes this generic combination of heap and value analyses. First of all, Section 2 introduces the minimal object-oriented language we deal with. Then we formalize a standard concrete and a split domain and semantics in Section 3. Section 4 formalizes and proves the soundness of the abstract domain and semantics. Section 5 shows how to plug in our framework pointer and shape analyses, as well as numerical domains. In this way, we prove that our approach is generic enough to support some of the most common heap and value analyses. Finally, Section 6 discusses the related work, while Section 7 concludes.

Notation: In this paper, we will denote by $\rightarrow_{\mathsf{A}}$ a small-step transition semantics on the domain A, and by $\rightarrow_{\wp(\mathsf{A})}$ the pointwise application of $\rightarrow_{\mathsf{A}}$ to set of states in A. Formally, $\langle \mathtt{st}, \mathsf{A}_1 \rangle \rightarrow_{\wp(\mathsf{A})} \{\mathsf{a}' : \exists \mathsf{a} \in \mathsf{A}_1 : \langle \mathtt{st}, \mathsf{a} \rangle \rightarrow_{\mathsf{A}} \mathsf{a}'\}$ where $\mathsf{A}_1 \subseteq \mathsf{A}$. With an abuse of notation, we will denote by π_n the projection of a set of tuples (with at least n components) to the set containing the n-th component of each single tuple in the given set. Our approach is based on the abstract interpretation theory [9,10]. We will denote concrete sets and elements by C and c, respectively, and abstract sets and elements by $\overline{\mathsf{A}}$ and $\overline{\mathsf{a}}$, respectively. In addition, $\gamma_{\overline{\mathsf{A}}}$ will denote the concretization function of the abstract domain $\overline{\mathsf{A}}$.

2 Language

A program consists of a control flow graph of basic blocks. Each basic block consists of a sequence of statements. Different blocks are connected through edges that optionally contain a Boolean condition to represent conditional jumps.

[2] For the sake of simplicity, we assume that `l` is acyclic containing at least two elements.

$$\frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathsf{s}(\mathsf{e}(\mathtt{y}), \mathtt{f})]}{\langle \mathtt{x} = \mathtt{y.f}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_\Sigma (\mathsf{e}', \mathsf{s})} \qquad \frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathsf{e}(\mathtt{y})]}{\langle \mathtt{x} = \mathtt{y}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_\Sigma (\mathsf{e}', \mathsf{s})}$$

$$\frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathit{alloc}(\mathtt{C}, (\mathsf{e}, \mathsf{s}))]}{\langle \mathtt{x} = \mathtt{new}\ \mathtt{C}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_\Sigma (\mathsf{e}', \mathsf{s})} \qquad \frac{\mathsf{s}' = \mathsf{s}[(\mathsf{e}(\mathtt{x}), \mathtt{f}) \mapsto \mathsf{e}(\mathtt{y})]}{\langle \mathtt{x.f} = \mathtt{y}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_\Sigma (\mathsf{e}, \mathsf{s}')}$$

$$\frac{\mathsf{s}' = \mathsf{s}[(\mathsf{e}(\mathtt{x}), \mathtt{f}) \mapsto \mathit{eval}(\mathtt{vexp}, (\mathsf{e}, \mathsf{s}))]}{\langle \mathtt{x.f} = \mathtt{vexp}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_\Sigma (\mathsf{e}, \mathsf{s}')} \qquad \frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathit{eval}(\mathtt{vexp}, (\mathsf{e}, \mathsf{s}))]}{\langle \mathtt{x} = \mathtt{vexp}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_\Sigma (\mathsf{e}', \mathsf{s})}$$

Fig. 4. The concrete semantics $\rightarrow_\Sigma$

Table 1. Expressions and statements

$$\begin{array}{l} \mathtt{rexp} ::= \mathtt{x} \mid \mathtt{x.f} \mid \mathtt{new}\ \mathtt{C} \\ \mathtt{vexp} ::= \mathtt{x} \mid \mathtt{x.f} \mid \mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2} \\ \mathtt{op} ::= + \mid - \mid * \mid \cdots \\ \mathtt{st} ::= \mathtt{x} = \mathtt{rexp} \mid \mathtt{x.f} = \mathtt{y} \mid \\ \qquad \mid \mathtt{x} = \mathtt{vexp} \mid \mathtt{x.f} = \mathtt{vexp} \end{array}$$

For the sake of simplicity, we focus our attention on the statements of the minimal object-oriented language defined in Table 1. This supports assignments to variables and fields, and it distinguishes among value and reference expressions. Reference expressions can be variable identifiers, field accesses, and object creations. Value expressions can be variables, field accesses, or binary combinations of value expressions through an (e.g., arithmetic) operator. Therefore, we assume that we can distinguish between value (that is, `vexp` returning values of native types like `int` and `double` in Java) and reference expressions (that is, `rexp`). Note that in Table 1 `y` represents a variable of reference type, and $\mathtt{C} \in \mathsf{Class}$ where Class denotes the set of the classes of the object-oriented program that can be instantiated.

3 Concrete Domain and Semantics

In this section we first introduce a standard domain (Σ) and semantics ($\rightarrow_\Sigma$) of object-oriented programs, and we abstract it with a split domain (Σ_{Split}) and semantics ($\rightarrow_{\mathsf{Split}}$) proving the soundness of our approach.

3.1 Standard Domain and Semantics

First of all, we partition the content of variables and heap locations into values (Val) and references (Ref). As usual in object-oriented programming languages, a state of the execution is composed by an environment (that relates local variables to references or values, $\mathsf{Env} : \mathsf{Var} \rightarrow (\mathsf{Ref} \cup \mathsf{Val})$) and a store (that relates locations to references or values, $\mathsf{Store} : (\mathsf{Ref} \times \mathsf{Field}) \rightarrow (\mathsf{Ref} \cup \mathsf{Val})$). A concrete state is defined by $\Sigma = \mathsf{Env} \times \mathsf{Store}$. The lattice structure is given by $\langle \wp(\Sigma), \subseteq \rangle$.

Semantics. Figure 4 defines a standard concrete small step semantics $\rightarrow_\Sigma$, while Figure 5 defines a standard concrete evaluation of value expressions. We assume that a function *alloc* : $(\mathsf{Class} \times \Sigma) \rightarrow \mathsf{Ref}$ is given. This allocates an object instance of the given class, and returns the reference pointing to it.

$$\begin{aligned}&\mathit{eval} : (\mathtt{vexp} \times \Sigma) \rightarrow \mathsf{Val}\\ &\mathit{eval}(\mathtt{x}, (\mathsf{e}, \mathsf{s})) = \mathsf{e}(\mathtt{x})\\ &\mathit{eval}(\mathtt{x.f}, (\mathsf{e}, \mathsf{s})) = \mathsf{s}(\mathsf{e}(\mathtt{x}), \mathtt{f})\\ &\mathit{eval}(\mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}, (\mathsf{e}, \mathsf{s})) = \mathit{eval}(\mathtt{vexp1}, (\mathsf{e}, \mathsf{s})) < \mathtt{op} > \mathit{eval}(\mathtt{vexp2}, (\mathsf{e}, \mathsf{s}))\end{aligned}$$

Fig. 5. The concrete expression evaluation

$$\begin{aligned}&\mathit{eval}_{\mathsf{Split}} : (\mathtt{vexp}' \times \Sigma_{\mathsf{Val}}) \rightarrow \mathsf{Val}\\ &\mathit{eval}_{\mathsf{Split}}(\mathtt{x}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) = \mathsf{e}_{\mathsf{Val}}(\mathtt{x})\\ &\mathit{eval}_{\mathsf{Split}}(< \mathtt{r} > .\mathtt{f}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) = \mathsf{s}_{\mathsf{Val}}(\mathsf{r}, \mathtt{f})\\ &\mathit{eval}_{\mathsf{Split}}(\mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) =\\ &\qquad = \mathit{eval}_{\mathsf{Split}}(\mathtt{vexp1}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) < \mathtt{op} > \mathit{eval}_{\mathsf{Split}}(\mathtt{vexp2}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}))\end{aligned}$$

Fig. 6. The split expression evaluation

3.2 Split Domain

We split the concrete domain and semantics between the portion dealing with values ($\mathsf{Env}_{\mathsf{Val}} : \mathsf{Var} \rightarrow \mathsf{Val}$, $\mathsf{Store}_{\mathsf{Val}} : (\mathsf{Ref} \times \mathsf{Field}) \rightarrow \mathsf{Val}$, and $\Sigma_{\mathsf{Val}} = \mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\mathsf{Val}}$), and the portion dealing with references ($\mathsf{Env}_{\mathsf{Ref}} : \mathsf{Var} \rightarrow \mathsf{Ref}$, $\mathsf{Store}_{\mathsf{Ref}} : (\mathsf{Ref} \times \mathsf{Field}) \rightarrow \mathsf{Ref}$, and $\Sigma_{\mathsf{Ref}} = \mathsf{Env}_{\mathsf{Ref}} \times \mathsf{Store}_{\mathsf{Ref}}$). A state is then the Cartesian product of these two components ($\Sigma_{\mathsf{Split}} = \Sigma_{\mathsf{Ref}} \times \Sigma_{\mathsf{Val}}$). Like the concrete domain, the lattice structure is given by set of elements, that is, $\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$.

Soundness. To prove the soundness of $\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$ with respect to $\langle \wp(\Sigma), \subseteq \rangle$, we have to formalize the concretization $\gamma_{\mathsf{Split}} : \wp(\Sigma_{\mathsf{Split}}) \rightarrow \wp(\Sigma)$ that defines how states in Σ_{Split} are mapped into states in Σ. Intuitively, this consists in the pointwise set union of the two parts of split states. Formally, $\gamma_{\mathsf{Split}}(\mathsf{T}) = \{(\mathsf{e}_v \cup \mathsf{e}_h, \mathsf{s}_v \cup \mathsf{s}_h) : ((\mathsf{e}_h, \mathsf{s}_h), (\mathsf{e}_v, \mathsf{s}_v)) \in \mathsf{T}\}$. Note that, since in the definition of the language in Section 2 we assumed that we can distinguish among value and reference expressions (and in particular local variables and field accesses), the domains of e_v and e_h do not overlap. The same considerations apply to s_v and s_h.

Then, we have that $\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$ is a sound approximation of $\langle \wp(\Sigma), \subseteq \rangle$, that is, they form a Galois connection.

Semantics. Figure 6 defines the evaluation of expressions in $\mathtt{vexp}'$, where $\mathtt{vexp}' ::= \mathtt{x} \,|\, < \mathtt{r} > .\mathtt{f} \,|\, \mathtt{vexp1}' < \mathtt{op} > \mathtt{vexp2}'$ with $\mathtt{r} \in \mathsf{Ref}$. The main difference w.r.t. the concrete expression evaluation defined by Figure 5 is that it deals only with the value portion of the heap state. This is possible since $\mathtt{vexp}'$ contains

$$\frac{\mathsf{s}_{\mathsf{Val}}' = \mathsf{s}_{\mathsf{Val}}[(\mathtt{r}, \mathtt{f}) \mapsto \mathit{eval}_{\mathsf{Split}}(\mathtt{vexp}', (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}))]}{\langle < \mathtt{r} > .\mathtt{f} = \mathtt{vexp}', (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}) \rangle \rightarrow_{\mathsf{Val}} (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}')} \quad \frac{\mathsf{e}_{\mathsf{Val}}' = \mathsf{e}_{\mathsf{Val}}[\mathtt{x} \mapsto \mathit{eval}_{\mathsf{Split}}(\mathtt{vexp}', (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}))]}{\langle \mathtt{x} = \mathtt{vexp}', (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}) \rangle \rightarrow_{\mathsf{Val}} (\mathsf{e}_{\mathsf{Val}}', \mathsf{s}_{\mathsf{Val}})}$$

Fig. 7. The semantics of the value part

$$\frac{\mathsf{e}'_{\mathsf{Ref}} = \mathsf{e}_{\mathsf{Ref}}[\mathtt{x} \mapsto alloc(\mathtt{C}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}}))]}{\langle \mathtt{x} = \mathtt{new\ C}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}'_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})} \qquad \frac{\mathsf{e}'_{\mathsf{Ref}} = \mathsf{e}_{\mathsf{Ref}}[\mathtt{x} \mapsto \mathsf{s}_{\mathsf{Ref}}(\mathsf{e}_{\mathsf{Ref}}(\mathtt{y}), \mathtt{f})]}{\langle \mathtt{x} = \mathtt{y.f}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}'_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})}$$

$$\frac{\mathsf{s}'_{\mathsf{Ref}} = \mathsf{s}_{\mathsf{Ref}}[(\mathsf{e}_{\mathsf{Ref}}(\mathtt{x}), \mathtt{f}) \mapsto \mathsf{e}_{\mathsf{Ref}}(\mathtt{y})]}{\langle \mathtt{x.f} = \mathtt{y}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}'_{\mathsf{Ref}})} \qquad \frac{\mathsf{e}'_{\mathsf{Ref}} = \mathsf{e}_{\mathsf{Ref}}[\mathtt{x} \mapsto \mathsf{e}_{\mathsf{Ref}}(\mathtt{y})]}{\langle \mathtt{x} = \mathtt{y}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}'_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})}$$

Fig. 8. The semantics of the heap part

$$\frac{\langle \mathtt{x} = \mathtt{rexp}, \sigma_{\mathsf{Ref}}\rangle \rightarrow_{\mathsf{Ref}} \sigma'_{\mathsf{Ref}}}{\langle \mathtt{x} = \mathtt{rexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma'_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})} \qquad \frac{\langle \mathtt{x.f} = \mathtt{y}, \sigma_{\mathsf{Ref}}\rangle \rightarrow_{\mathsf{Ref}} \sigma'_{\mathsf{Ref}}}{\langle \mathtt{x.f} = \mathtt{y}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma'_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})}$$

$$\frac{\langle \mathbb{R}[\![\mathtt{x.f}, \sigma_{\mathsf{Ref}}]\!] = \mathbb{R}[\![\mathtt{vexp}, \sigma_{\mathsf{Ref}}]\!], \sigma_{\mathsf{Val}}\rangle \rightarrow_{\mathsf{Val}} \sigma'_{\mathsf{Val}}}{\langle \mathtt{x.f} = \mathtt{vexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma_{\mathsf{Ref}}, \sigma'_{\mathsf{Val}})} \qquad \frac{\langle \mathtt{x} = \mathbb{R}[\![\mathtt{vexp}, \sigma_{\mathsf{Ref}}]\!], \sigma_{\mathsf{Val}}\rangle \rightarrow_{\mathsf{Val}} \sigma'_{\mathsf{Val}}}{\langle \mathtt{x} = \mathtt{vexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma_{\mathsf{Ref}}, \sigma'_{\mathsf{Val}})}$$

Fig. 9. The semantics of the split domain

a reference instead of a local variable when accessing a field with statement $<\mathtt{r}>.\mathtt{f}$. Then, we have that $\rightarrow_{\mathsf{Split}}$ is a sound approximation of $\rightarrow_{\Sigma}$.

The small-step semantics $\rightarrow_{\mathsf{Split}}$ over Σ_{Split} is formalized by Figure 9. It mainly applies the proper semantics of the value (Figure 7) or the heap (Figure 8) part of the state by looking to the statement. The only noticeable difference appears when we deal with statements requiring both value and heap state (namely, $\mathtt{x} = \mathtt{vexp}$ and $\mathtt{x.f} = \mathtt{vexp}$). In these cases, we preprocess $\mathtt{vexp}$ and $\mathtt{x.f}$ with the following function $\mathbb{R}$:

$\mathbb{R} : (\mathtt{vexp} \times \Sigma_{\mathsf{Ref}}) \rightarrow \mathtt{vexp}'$
$\mathbb{R}[\![\mathtt{x}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] = \mathtt{x}$
$\mathbb{R}[\![\mathtt{x.f}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] = <\mathsf{e}_{\mathsf{Ref}}(\mathtt{x})>.\mathtt{f}$
$\mathbb{R}[\![\mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] =$
$\qquad = \mathbb{R}[\![\mathtt{vexp1}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] < \mathtt{op} > \mathbb{R}[\![\mathtt{vexp2}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!]$

This function replaces in a value expression the local variable $\mathtt{x}$ in a field access $\mathtt{x.f}$ with the reference $\mathtt{r}$ pointed by $\mathtt{x}$ in the reference environment. This step is necessary to allow the evaluation of value expressions to perform without any knowledge of the actual state of the reference part.

4 Abstract Domain and Semantics

The reference (Σ_{Ref}) and the value (Σ_{Val}) part of the split domain are approximated by a given heap ($\overline{\mathsf{H}}$) and value ($\overline{\mathsf{V}}$) analysis, respectively. In addition, the heap analysis defines a set of heap identifiers $\overline{\mathsf{HId}}$ which aims at abstracting concrete locations, and the value analysis tracks information over these identifiers like it does over variable identifiers. Since we want to allow the heap analysis to merge and materialize heap identifiers, we have to communicate these changes to the value analysis through substitutions. In this Section, we formalize and prove the soundness of this framework.

4.1 Abstract Domain

We assume that a value analysis $\overline{\mathsf{V}}$ and a heap analysis $\overline{\mathsf{H}}$ are provided with lattice operators ($\langle\overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}, \sqcup_{\overline{\mathsf{V}}}, \sqcap_{\overline{\mathsf{V}}}\rangle$ and $\langle\overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}} \sqcup_{\overline{\mathsf{H}}}, \sqcap_{\overline{\mathsf{H}}}\rangle$, respectively). In addition, they provide the widening operators $\nabla_{\overline{\mathsf{V}}}$ and $\nabla_{\overline{\mathsf{H}}}$, respectively. The states $\overline{\Sigma}$ of our abstract domain are composed by a heap and a value state ($\overline{\Sigma} = \overline{\mathsf{H}} \times \overline{\mathsf{V}}$). Then a state of our analysis is the Cartesian product of $\overline{\mathsf{H}}$ and $\overline{\mathsf{V}}$ denoted by $\langle\overline{\Sigma}, \sqsubseteq_{\overline{\Sigma}}, \sqcup_{\overline{\Sigma}}, \sqcap_{\overline{\Sigma}}\rangle$.

4.2 Concretization Function

We assume that the heap analysis defines a finite set of heap identifiers $\overline{\mathsf{HId}}$. In addition, it provides a function $\overline{\mathit{heapId}} : \overline{\mathsf{H}} \to \wp(\overline{\mathsf{HId}})$ that returns the set of heap identifiers contained in a given abstract heap. The value analysis tracks information on variables as well as heap identifiers. Therefore, the concretization of the value analysis produces (i) environments in $\mathsf{Env}_{\mathsf{Val}}$, and (ii) stores with abstract heap identifiers instead of concrete locations since the value analysis alone cannot concretize heap identifiers ($\mathsf{Store}_{\overline{\mathsf{HId}}} : \overline{\mathsf{HId}} \to \wp(\mathsf{Val})$). Note that we have a set of concrete values as codomain since a single heap identifier, representing a summary node, may concretize into many concrete references that could have different values in the same store's concretization. For instance, a list of positive values may be approximated by a single heap identifier `u`. Imagine that we are dealing with a particular concretization of this list containing two elements. These two elements are not necessarily equal, so for instance the value analysis could tell us that `u.f` concretizes to $\{1, 2\}$ (e.g., to represent a list $[1; 2]$).

To concretize a store in $\mathsf{Store}_{\overline{\mathsf{HId}}}$ to Store, we need that the heap analysis provides the concretization of heap identifiers. Therefore, the heap concretization has to provide a function $\gamma_{\overline{\mathsf{HId}}}$ that relates each heap identifier to a set of concrete locations ($\overline{\mathsf{HId}} \to \wp(\mathsf{Ref} \times \mathsf{Field})$). Also in this case, we need to have a set of concrete locations as codomain in order to support summary nodes.

In this way, the heap analysis can track the shape of the heap, and represent symbolically nodes using heap identifiers. When it concretizes, the concrete values of references in one concrete store transform the shape into a concrete memory state. In this scenario, the heap identifiers' concretization is the component that tells us how we go from the shape to the concrete references. Note that a single shape could concretize to a (possibly infinite) set of concrete stores, since there are infinitely many possible reference values for the heap identifiers, and the heap identifiers' concretization is specific for one concrete store.

Formally, the heap concretization $\gamma_{\overline{\mathsf{H}}}$ returns a set of pairs containing a concrete store, and a concretization of heap identifiers ($\gamma_{\overline{\mathsf{H}}} : \overline{\mathsf{H}} \to \wp(\Sigma_{\mathsf{Ref}} \times (\overline{\mathsf{HId}} \to \wp(\mathsf{Ref} \times \mathsf{Field})))$). The mapping of heap identifiers to sets of concrete locations is necessary to concretize later value stores.

We assume that the heap and value analyses are sound, that is, they form a Galois connection. Formally, $\langle \wp(\mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\overline{\mathsf{HId}}}), \subseteq\rangle \xleftrightarrow[\alpha_{\overline{\mathsf{V}}}]{\gamma_{\overline{\mathsf{V}}}} \langle \overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}\rangle$ and $\langle \wp(\Sigma_{\mathsf{Ref}}), \subseteq \rangle \xleftrightarrow[\alpha_{\overline{\mathsf{H}}}]{\pi_1 \circ \gamma_{\overline{\mathsf{H}}}} \langle \overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}}\rangle$. In addition, we assume that $\gamma_{\overline{\mathsf{V}}}$ and $\pi_1(\gamma_{\overline{\mathsf{H}}})$ are complete meet-morphisms.

We are now in position to combine the heap and value concretizations to concretize abstract states in $\overline{\Sigma}$ to $\wp(\Sigma_{\mathsf{Split}})$. What is still missing is the concretization of $\mathsf{Store}_{\overline{\mathsf{HId}}}$ to Store. Intuitively, the resulting stores should relate a location $(\mathsf{r}, \mathtt{f})$ with the values related to a heap identifier that is concretized into $(\mathsf{r}, \mathtt{f})$. Formally, we define $\gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}} : (\mathsf{Store}_{\overline{\mathsf{HId}}} \times (\overline{\mathsf{HId}} \to \wp(\mathsf{Ref} \times \mathsf{Field}))) \to \wp(\mathsf{Store}_{\mathsf{Val}})$ as follows:

$$\gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}(\mathsf{s}, \gamma_{\overline{\mathsf{HId}}}) = \{[(\mathsf{r}, \mathtt{f}) \mapsto \mathsf{v}] : \mathsf{i} \in dom(\gamma_{\overline{\mathsf{HId}}}) \wedge (\mathsf{r}, \mathtt{f}) \in \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}) \wedge \mathsf{v} \in \mathsf{s}(\mathsf{i})\}$$

Finally, the concretization of abstract states $\gamma_{\overline{\Sigma}} : \overline{\Sigma} \to \wp(\Sigma_{\mathsf{Split}})$ is defined by:

$$\gamma_{\overline{\Sigma}}(\overline{\mathsf{v}}, \overline{\mathsf{h}}) = \{(\sigma_H, (\mathsf{e}_v, \mathsf{s}'_v)) : (\mathsf{e}_v, \mathsf{s}_v) \in \gamma_{\overline{\mathsf{V}}}(\overline{\mathsf{v}}) \wedge (\sigma_H, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}) \wedge \\ \mathsf{s}'_v \in \gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}(\mathsf{s}_v, \gamma_{\overline{\mathsf{HId}}})\}$$

Running Example. Consider now the motivating example of Figure 1 with the list passed by `client1`, and the abstract state depicted in the right part of Figure 2. The numerical domain concretizes the store (s_v in the definition of $\gamma_{\overline{\Sigma}}$) to $\{[(\mathtt{p1}, \mathtt{f}) \mapsto \{1\}, (\mathtt{p2}, \mathtt{f}) \mapsto \{2\}]\}$ while the value environment e_v is empty since there is no local value variable. Instead, $\gamma_{\overline{\mathsf{H}}}$ may concretize to many heaps with different $\gamma_{\overline{\mathsf{HId}}}$. For the sake of simplicity, let us focus on the case in which $\gamma_{\overline{\mathsf{HId}}} = [(\mathtt{p1}, \mathtt{f}) \mapsto \{(\#1, \mathtt{f})\}, (\mathtt{p2}, \mathtt{f}) \mapsto \{(\#2, \mathtt{f})\}]$. Then $\gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}$ returns the numerical store $s'_v = [(\#1, \mathtt{f}) \mapsto 1, (\#2, \mathtt{f}) \mapsto 2]$, that is, it substitutes $(\mathtt{p1}, \mathtt{f})$ and $(\mathtt{p2}, \mathtt{f})$ with $(\#1, \mathtt{f})$ and $(\#2, \mathtt{f})$ in s_v, respectively.

Soundness. We need to assume some conditions on how heap identifiers are concretized in order to prove the soundness of the analysis.

C1 $\gamma_{\overline{\mathsf{HId}}}$ has to define the concretization of all the heap identifiers contained in the concretized heap. Formally, $\forall \gamma_{\overline{\mathsf{HId}}} \in \pi_2(\gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}})) : dom(\gamma_{\overline{\mathsf{HId}}}) = \overline{heapId}(\overline{\mathsf{h}})$.

C2 If a heap identifier is not in all the states we are intersecting, then it will not be part of the results of the intersection, since through the greatest lower bound we are taking only the part that is common among all the states we are intersecting. Formally, $\forall \overline{\mathsf{H}}_1 \subseteq \overline{\mathsf{H}} : \overline{heapId}(\bigsqcap_{\overline{\mathsf{h}} \in \overline{\mathsf{H}}_1} \overline{\mathsf{h}}) = \bigcap_{\overline{\mathsf{h}} \in \overline{\mathsf{H}}_1} \overline{heapId}(\overline{\mathsf{h}})$.

C3 Different heap identifiers represent different portions of the heap. Formally, $\forall \mathsf{i}_1, \mathsf{i}_2 \in dom(\gamma_{\overline{\mathsf{HId}}}) : \mathsf{i}_1 \neq \mathsf{i}_2 \Rightarrow \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}_1) \cap \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}_2) = \emptyset$.

C4 When we intersect a set of abstract heaps, the heap identifiers' concretization is the pointwise intersection of the heap identifiers' concretization of all the intersected states. Formally, $\forall (\sigma_H, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\bigsqcap_{\overline{\mathsf{h}} \in \overline{\mathsf{H}}_1} \overline{\mathsf{h}}), \forall (\sigma_H, \gamma^i_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}_i) : \overline{\mathsf{h}}_i \in \overline{\mathsf{H}}_1 \Rightarrow \gamma_{\overline{\mathsf{HId}}} = \lambda \mathsf{x}. \bigcap_{\overline{\mathsf{h}}_i \in \overline{\mathsf{H}}_1} \gamma^i_{\overline{\mathsf{HId}}}(\mathsf{x})$.

Condition **C1** is necessary since otherwise the value analysis could track information on heap identifiers that it does not know how to concretize. Condition **C3** is a rather standard assumption over abstract nodes in heap analysis (e.g., in shape analysis [30]), and it states that different identifiers represents different portions of the heap. In this way, the assignment of an identifier is guaranteed not to affect other identifiers. Condition **C2** and **C4** are both necessary to prove that $\gamma_{\overline{\Sigma}}$ is meet-preserving. This is a fundamental property for Galois connections induced by a glb-preserving concretization function (e.g., see Proposition 7 of [11]). Intuitively, they correspond to the requirement that $\pi_1(\gamma_{\overline{\mathsf{H}}})$ and $\gamma_{\overline{\mathsf{V}}}$ are complete meet-morphism applied to $\pi_2(\gamma_{\overline{\mathsf{H}}})$.

Theorem 1 ($\langle\overline{\Sigma}, \sqsubseteq\rangle$ is a sound approximation of $\langle\wp(\Sigma_{\mathsf{Split}}), \subseteq\rangle$) $\langle\wp(\Sigma_{\mathsf{Split}}), \subseteq\rangle \underset{\alpha_{\overline{\Sigma}}}{\overset{\gamma_{\overline{\Sigma}}}{\leftrightarrows}} \langle\overline{\Sigma}, \sqsubseteq\rangle$ *where* $\alpha_{\overline{\Sigma}} = \lambda\mathsf{X}.\sqcap_{\overline{\Sigma}}\{\overline{\mathsf{x}} : \mathsf{X} \subseteq \gamma_{\overline{\Sigma}}(\overline{\mathsf{x}})\}$.

Since Galois connections compose [9], we have that $\langle\wp(\Sigma), \subseteq\rangle \underset{\alpha_{\overline{\Sigma}} \circ \alpha_{\wp(\mathsf{Split})}}{\overset{\gamma_{\wp(\mathsf{Split})} \circ \gamma_{\overline{\Sigma}}}{\leftrightarrows}}$ $\langle\overline{\Sigma}, \sqsubseteq\rangle$ that is, $\overline{\Sigma}$ is sound with respect to the standard concrete domain $\wp(\Sigma)$.

4.3 Substitutions

Substitutions allow the heap analysis to freely manage heap identifiers when applying semantic operators. They are defined by $\overline{\mathsf{Sub}} : \wp(\overline{\mathsf{HId}}) \to \wp(\overline{\mathsf{HId}})$. The meaning of a relation in a substitution is that the identifiers in the domain are in the post-state, and they derive from the identifiers in the pre-state they are in relation with. For instance, $[\{\mathsf{id1}, \mathsf{id2}\} \mapsto \{\mathsf{id3}\}]$ represents that $\mathsf{id1}$ and $\mathsf{id2}$ are materialized from $\mathsf{id3}$, while $[\{\mathsf{id1}\} \mapsto \{\mathsf{id2}, \mathsf{id3}\}]$ means that $\mathsf{id2}$ and $\mathsf{id3}$ are merged into $\mathsf{id1}$.

In our notation, we will represent by $\rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}}$ that the heap semantic operator $\rightarrow_{\overline{\mathsf{H}}}$ produced the substitution sub. Function $\overline{applySub} : (\overline{\mathsf{V}} \times \overline{\mathsf{Sub}}) \to \overline{\mathsf{V}}$ applies a substitution to a state of the value analysis, and it is defined by:

$$\begin{aligned}\overline{applySub}(\overline{\mathsf{v}}, \mathsf{sub}) = \overline{\mathsf{v}}_n \text{ where } \mathsf{sub} = [\mathsf{I}_1 \mapsto \mathsf{I}'_1, ..., \mathsf{I}_n \mapsto \mathsf{I}'_n], \overline{\mathsf{v}}_0 = \overline{\mathsf{v}} \wedge \\ \forall j \in [1..n] : \overline{\mathsf{v}}_j = \sqcup_{\mathsf{i}' \in \mathsf{I}'_j}\{\overline{\mathsf{v}}'_n : \mathsf{I}_j = \{\mathsf{i}_1, \cdots, \mathsf{i}_n\}, \overline{\mathsf{v}}'_0 = \overline{\mathsf{v}}_{j-1}, \\ \forall \mathsf{k} \in [1..n] : \langle \mathsf{i}_{\mathsf{k}} = \mathsf{i}', \overline{\mathsf{v}}'_{k-1}\rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}'_k\}\end{aligned}$$

The intuition behind $\overline{applySub}$ is that, given a substitution sub, each single *replacement* $\mathsf{I}_j \mapsto \mathsf{I}'_j \in \mathsf{sub}$ represents that the identifiers in I'_j are substituted by I_j. Therefore, each identifier $\mathsf{i}' \in \mathsf{I}'_j$ is assigned to each identifier $\mathsf{i} \in \mathsf{I}_j$.

Soundness. We expect that the substitution produced by a heap semantic operator is coherent with respect to the modifications of the heap identifiers that have been induced by such operator. This means that, if $\langle \mathtt{st}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}'$ and $[\mathsf{I} \mapsto \mathsf{I}'] \in \mathsf{sub}$, the concrete locations represented by I' in the pre-state corresponds to what is represented by I in the post-state. This correspondence is bound to heap concretization that are related through the concrete heap semantics $\rightarrow_{\mathsf{Ref}}$. This concept is formalized by the following proposition.

Proposition 1 (Soundness of the substitution). *Let* $\overline{\mathsf{h}} \in \overline{\mathsf{H}}$ *be a state of the heap analysis such that* $\langle \mathtt{st}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}'$.

A substitution is sound iff $\forall(\mathsf{h}, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}) : \langle \mathtt{st}, \mathsf{h}\rangle \rightarrow_{\mathsf{Ref}} \mathsf{h}', (\mathsf{h}', \gamma'_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}')$ *we have that* $\gamma'_{\overline{\mathsf{HId}}} = \gamma_{\overline{\mathsf{HId}}}[\mathsf{i} \mapsto \mathsf{I}' : \mathsf{I}' \subseteq \bigcup_{\mathsf{i}_1 \in \mathsf{sub}(\mathsf{I})} \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}_1) \wedge \exists \mathsf{I} \in dom(\mathsf{sub}) : \mathsf{i} \in \mathsf{I}]$ *and* $\forall \mathsf{I} \in dom(\mathsf{sub}) : \bigcup_{\mathsf{i} \in \mathsf{I}} \gamma'_{\overline{\mathsf{HId}}}(\mathsf{i}) = \bigcup_{\mathsf{i}' \in \mathsf{sub}(\mathsf{I})} \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}')$.

Intuitively, the substitution univocally establishes how heap identifiers' concretization is affected by the heap semantic operator. In addition, since a substitution represents how heap identifiers are modified in a single step, we do not want that different replacements in the same substitution overlap. The intuition is that a set of heap identifiers can be substituted by another set, but during one single substitution the same heap identifiers cannot be in many replacements. For instance, imagine that we have the substitution $[\{\texttt{id1}\} \mapsto \{\texttt{id2}\}, \{\texttt{id2}\} \mapsto \{\texttt{id3}\}]$. In this case, it is not clear what is represented. In particular, we could have that (i) `id1` is replaced by `id2` that is then replaced by `id3`, or (ii) `id2` is replaced by `id3` and `id1` is replaced by `id2`. To avoid this ambiguity, we do not allow this scenario. Nevertheless, the effects of overlapping substitutions (like the one we sketched) can be obtained by a sequence of non-overlapping substitutions that disambiguate the semantics. For instance, (i) corresponds to $[\{\texttt{id1}\} \mapsto \{\texttt{id2}\}]$ followed by $[\{\texttt{id2}\} \mapsto \{\texttt{id3}\}]$, while (ii) corresponds to $[\{\texttt{id2}\} \mapsto \{\texttt{id3}\}]$ followed by $[\{\texttt{id1}\} \mapsto \{\texttt{id2}\}]$.

Proposition 2 (Non-overlapping replacements). *We assume that single replacements in the same substitution do not overlap. Formally,* $\forall \mathsf{l} \in dom(\mathsf{sub})$, $\forall \mathsf{l}' \in dom(\mathsf{sub}) \setminus \{\mathsf{l}\} : \mathsf{l} \cap \mathsf{l}' = \emptyset \wedge \mathsf{sub}(\mathsf{l}) \cap \mathsf{sub}(\mathsf{l}') = \emptyset$.

Running Example. Consider again the program in Figure 1, and suppose to have a transition like the one depicted in Figure 3. The materialization of `u2` produces the replacement $[\{\texttt{u1.f}, \texttt{u2.f}\} \mapsto \{\texttt{u1.f}\}]$, and its application to the numerical state produces $[\texttt{u1.f} \mapsto [0..+\infty], \texttt{u2.f} \mapsto [0..+\infty]]$.

This replacement satisfies the soundness conditions of the substitution. Intuitively, the two heap identifiers `u1.f` and `u2.f` in the post state corresponds to `u1.f` in the heap state. Therefore, given a particular concrete state, the concretization of the heap identifiers of `u1.f` and `u2.f` in the post-state corresponds to the concretization of `u1.f` in the pre-state, that is exactly what is stated by Proposition 1. In addition, the substitution does not overlap, and therefore it satisfies Proposition 2.

4.4 Semantics

In our split domain, we assumed that the value part Σ_{Val} takes care of statements about values, while the heap part Σ_{Ref} defines the semantics of statements dealing with the heap. Nevertheless, we needed the heap state to replace field accesses with a reference and the accessed field in value expressions, and when assigning a value to a location. Similarly, we will have to replace field accesses with heap identifiers when defining the abstract semantics.

Let us define $\overline{\mathsf{vexp}} ::= \texttt{x} \,|\, \texttt{i} \,|\, \texttt{vexp1} < \texttt{op} > \texttt{vexp2}$ where $\mathsf{i} \in \overline{\mathsf{HId}}$. $\overline{\mathsf{vexp}}$ is the abstract counterpart of $\texttt{vexp}'$. We assume that $\overline{\mathsf{V}}$ provides the semantics of value assignment $\langle \texttt{i} = \overline{\mathsf{vexp}}, \overline{\mathsf{v}} \rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}'$, and that $\overline{\mathsf{H}}$ provides (i) the semantics of field access $\langle \texttt{x.f}, \overline{\mathsf{h}} \rangle \rightarrow_{\overline{\mathsf{H}}} \overline{\mathsf{I}}$ (where $\overline{\mathsf{I}} \subseteq \overline{\mathsf{HId}}$ is the set of heap identifiers obtained by

$$\frac{\langle \mathtt{x.f} = \mathtt{y}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}'} \wedge \overline{applySub}(\overline{\mathsf{v}}, \mathsf{sub}) = \overline{\mathsf{v}}'}{\langle \mathtt{x.f} = \mathtt{y}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}', \overline{\mathsf{v}}')} \qquad \frac{\overline{\mathsf{v}}' = \bigsqcup_{\substack{\mathtt{i} \in \overline{\mathbb{R}}[\![\mathtt{x.f}, \overline{\mathsf{h}}]\!],\\ \overline{\mathsf{vexp}} \in \overline{\mathbb{R}}[\![\mathsf{vexp}, \overline{\mathsf{h}}]\!]}} \overline{\mathsf{v}}_1 : \langle \mathtt{i} = \overline{\mathsf{vexp}}, \overline{\mathsf{v}}\rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}_1}{\langle \mathtt{x.f} = \mathsf{vexp}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}, \overline{\mathsf{v}}')}$$

$$\frac{\langle \mathtt{x} = \mathtt{rexp}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}'} \wedge \overline{applySub}(\overline{\mathsf{v}}, \mathsf{sub}) = \overline{\mathsf{v}}'}{\langle \mathtt{x} = \mathtt{rexp}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}', \overline{\mathsf{v}}')} \qquad \frac{\overline{\mathsf{v}}' = \bigsqcup_{\overline{\mathsf{vexp}} \in \overline{\mathbb{R}}[\![\mathsf{vexp}, \overline{\mathsf{h}}]\!]} \overline{\mathsf{v}}_1 : \langle \mathtt{x} = \overline{\mathsf{vexp}}, \overline{\mathsf{v}}\rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}_1}{\langle \mathtt{x} = \mathsf{vexp}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}, \overline{\mathsf{v}}')}$$

Fig. 10. The abstract semantics $\rightarrow_{\overline{\Sigma}}$

$$\frac{}{\langle \mathtt{i} = \overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}})\rangle \rightarrow_{\mathsf{Val}'} (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}}[\mathtt{i} \mapsto eval'(\overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}}))])}$$

$$\frac{}{\langle \mathtt{x} = \overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}})\rangle \rightarrow_{\mathsf{Val}'} (e_{\mathsf{Val}}[\mathtt{x} \mapsto eval'(\overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}}))], s_{\overline{\mathsf{HId}}})}$$

Fig. 11. The semantics $\rightarrow_{\mathsf{Val}'}$

accessing x.f)[3], (ii) the semantics of local variable assignment $\langle \mathtt{x} = \mathtt{rexp}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}'$, and (iii) the semantics of field assignment $\langle \mathtt{x.f} = \mathtt{y}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}'$

The abstract semantics is defined by Figure 10. It relies on function $\overline{\mathbb{R}}$:
$\overline{\mathbb{R}} : (\mathsf{vexp} \times \overline{\mathsf{H}}) \rightarrow \wp(\overline{\mathsf{vexp}})$
$\overline{\mathbb{R}}[\![\mathtt{x}, \overline{\mathsf{h}}]\!] = \{\mathtt{x}\}$
$\overline{\mathbb{R}}[\![\mathtt{x.f}, \overline{\mathsf{h}}]\!] = \mathsf{I}$ where $\langle \mathtt{x.f}, \overline{\mathsf{h}}\rangle \rightarrow_{\overline{\mathsf{H}}} \mathsf{I}$
$$\overline{\mathbb{R}}[\![\mathsf{vexp1} < \mathsf{op} > \mathsf{vexp2}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})]\!] = \bigcup_{\substack{\overline{\mathsf{vexp1}} \in \overline{\mathbb{R}}[\![\mathsf{vexp1}, \overline{\mathsf{h}}]\!]\\ \overline{\mathsf{vexp2}} \in \overline{\mathbb{R}}[\![\mathsf{vexp2}, \overline{\mathsf{h}}]\!]}} \overline{\mathsf{vexp1}} < \mathsf{op} > \overline{\mathsf{vexp2}}$$

Similarly to $\mathbb{R}$, this function replaces each field access x.f with the heap identifier i that represents such field access in the current heap state. Since the heap analysis may return a set of heap identifiers when accessing a field (e.g., because it may track that a local variable could point to two different abstract heap nodes), $\overline{\mathbb{R}}$ returns a set of possible value expressions in $\overline{\mathsf{vexp}}$.

Running Example. Suppose to analyze the statement it = l at line 3 of the motivating example of Figure 1 obtaining a transition as depicted in Figure 3. The analysis materializes the node pointed by it, and it produces a substitution as discussed in Section 4.3. The definition of the semantics $\rightarrow_{\overline{\Sigma}}$ simply applies this substitution to the value analysis after the heap semantics.

[3] For the sake of simplicity, we assume that field accesses do not produce any substitution nor they modify the abstract heap state. Anyway, this does not limit the expressiveness of our approach, since we may obtain this substitution and a new heap state by simulating this statement by assigning a field access to a local variable, and then by replacing the field access with this local variable in the expression or assignment containing x.f.

$$
\begin{aligned}
&eval' : (\overline{\mathsf{vexp}} \times (\mathsf{Env_{Val}} \times \mathsf{Store}_{\overline{\mathsf{HId}}})) \to \mathsf{Val}\\
&eval'(\mathtt{x}, (\mathsf{e_{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) = \mathsf{e_{Val}}(\mathtt{x})\\
&eval'(\mathtt{i}, (\mathsf{e_{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) = \mathsf{s}_{\overline{\mathsf{HId}}}(\mathtt{i})\\
&eval'(\overline{\mathsf{vexp1}} < \mathsf{op} > \overline{\mathsf{vexp2}}, (\mathsf{e_{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) =\\
&\qquad\qquad = eval'(\overline{\mathtt{vexp1}}, (\mathsf{e_{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) < \mathsf{op} > eval'(\overline{\mathtt{vexp2}}, (\mathsf{e_{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}}))
\end{aligned}
$$

Fig. 12. The expression evaluation *eval'*

Soundness. Before establishing the soundness conditions of the heap and the value analyses, we need to introduce the semantics $\to_{\mathsf{Val}'}$ that defines the semantics of statements dealing with $\overline{\mathtt{vexp}}$ on $\mathsf{Env_{Val}} \times \mathsf{Store}_{\overline{\mathsf{HId}}}$. This is formalized by Figure 11 and 12. Then, we assume that the semantics of the value and the heap analysis are both sound.

Proposition 3 (Soundness of the value semantics). *We assume that the semantic operator provided by the value analysis is sound. Formally,*
$\forall \overline{\mathsf{v}} \in \overline{\mathsf{V}}, \langle \mathtt{st}, \overline{\mathsf{v}} \rangle \to_{\overline{\mathsf{V}}} \overline{\mathsf{v}}', \langle \mathtt{st}, \gamma_{\overline{\mathsf{V}}}(\overline{\mathsf{v}}) \rangle \to_{\wp(\mathsf{Val}')} \mathsf{V}' \Rightarrow \mathsf{V}' \subseteq \gamma_{\overline{\mathsf{V}}}(\overline{\mathsf{v}}')$
where $\mathtt{st} \in \{\mathtt{x} = \mathtt{vexp}, \mathtt{x.f} = \mathtt{vexp}\}$.

Proposition 4 (Soundness of the heap semantics). *We assume that the semantic operators provided by the heap analysis are sound. Formally,*

- $\forall \overline{\mathsf{h}} \in \overline{\mathsf{H}}, \langle \mathtt{st}, \overline{\mathsf{h}} \rangle \to^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}', \langle \mathtt{st}, \pi_1(\gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}})) \rangle \to_{\wp(\mathsf{Ref})} \mathsf{H}' \Rightarrow \mathsf{H}' \subseteq \pi_1(\gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}'))$ *where* $\mathtt{st} \in \{\mathtt{x} = \mathtt{rexp}, \mathtt{x.f} = \mathtt{y}\}$*, and*
- $\forall \overline{\mathsf{h}} \in \overline{\mathsf{H}}, \langle \mathtt{x.f}, \overline{\mathsf{h}} \rangle \to_{\overline{\mathsf{H}}} \mathsf{I}, \forall ((\mathsf{e_{Ref}}, \mathsf{s_{Ref}}), \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}) \Rightarrow (\mathsf{e_{Ref}}(\mathtt{x}), \mathtt{f}) \in \bigcup_{\mathsf{i} \in \mathsf{I}} \gamma_{\overline{\mathsf{HId}}}(\mathsf{i})$.

Theorem 2 (Soundness of $\to_{\overline{\Sigma}}$). *Let* $(\overline{\mathsf{h}}, \overline{\mathsf{v}}) \in \overline{\Sigma}$ *and* $\mathtt{st} \in \mathsf{St}$ *be a set of initial states and a statement, respectively. Then*
$\langle \mathtt{st}, (\overline{\mathsf{h}}, \overline{\mathsf{v}}) \rangle \to_{\overline{\Sigma}} (\overline{\mathsf{h}}', \overline{\mathsf{v}}'), \langle \mathtt{st}, \gamma_{\overline{\Sigma}}(\overline{\mathsf{h}}, \overline{\mathsf{v}}) \rangle \to_{\wp(\mathsf{Split})} \mathsf{S} \Rightarrow \mathsf{S} \subseteq \gamma_{\overline{\Sigma}}(\overline{\mathsf{h}}', \overline{\mathsf{v}}')$

4.5 Reduction

In abstract interpretation, the reduced product [10] allows two analyses to exchange information through a reduce operator. This operator is represented by a function $\rho_{\overline{\Sigma}} : \overline{\Sigma} \to \overline{\Sigma}$ such that (i) $\rho_{\overline{\Sigma}}(\overline{\sigma}) \sqsubseteq_{\overline{\Sigma}} \overline{\sigma}$, and (ii) $\gamma_{\overline{\Sigma}}(\rho_{\overline{\Sigma}}(\overline{\sigma})) = \gamma_{\overline{\Sigma}}(\overline{\sigma})$. Since the reduce operator may change what is represented by heap identifiers, it may produce a substitution whose effects are propagated to $\overline{\mathsf{V}}$ through $\overline{applySub}$.

For instance, a numerical analysis could discover that a list contains 2 elements, while the heap analysis was unable to track this information (e.g., it tracks the shape depicted on the left part of Figure 3). The reduce operator may refine the heap state leading to a shape similar to the one depicted in the upper right part of Figure 2.

Thanks to the genericness of the approach we adopted, we allow one to arbitrarily refine the heap state with the information tracked by the value analysis. Nevertheless, this refinement has to be defined on specific instances of value and heap analyses.

4.6 Interface of the Value and the Heap Analysis

We now summarize the interface of the value and the heap analysis. First of all, we have some standard assumptions on sound abstract domains. In particular, both the analyses form a lattice ($\langle\overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}, \sqcup_{\overline{\mathsf{V}}}, \sqcap_{\overline{\mathsf{V}}}\rangle$ and $\langle\overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}} \sqcup_{\overline{\mathsf{H}}}, \sqcap_{\overline{\mathsf{H}}}\rangle$) and a Galois connection with the concrete domain. In addition, they provide sound semantic operators to assign and read heap locations.

We have then some specific requirements on the heap analysis. In particular, $\overline{\mathsf{H}}$ provides (i) a finite set of heap identifiers $\overline{\mathsf{HId}}$, (ii) a function $\overline{heapId} : \overline{\mathsf{H}} \rightarrow \wp(\overline{\mathsf{HId}})$ that, given a state, returns the set of heap identifiers contained in that state, (iii) a coherent concretization of them (Conditions **C1-4**), and (iv) coherent substitutions of heap identifiers (Proposition 1 and 2).

These components are necessary to allow our framework to combine the heap and the value analyses, and to formally prove its soundness.

5 Instances

In this Section, we show how to plug two heap (namely, pointer and shape) analyses, and existing numerical domains in our framework. In this way, we prove that our framework is expressive enough to be applied to the most common heap and value analyses.

5.1 Pointer Analysis PA

Pointer analysis [20] has been extensively studied. One of the most known results in this field is Andersen's flow-insensitive analysis[1]. This analysis has been extended in various ways [31]. In this Section, we propose a slight modification of Might *et al.*'s analysis [26]. In particular, our analysis is flow-sensitive, field-sensitive, and it does not deal with context-sensitivity since we did not support method calls in our language. Nevertheless, we expect that our approach can be straightforwardly extended to such scenario. We adopt a standard allocation site abstraction [1,12,13,29] to approximate dynamic locations in a finite way.

Domain. Heap identifiers are represented by pairs made by (i) program labels in Lab of the `new` statements that allocate memory, and (ii) field names ($\overline{\mathsf{HId}}_{\mathsf{PA}} = \mathsf{Lab} \times \mathsf{Field}$). Since the sets of program labels and of field names are both finite, $\overline{\mathsf{HId}}_{\mathsf{PA}}$ is finite as well. The abstract environment relates each variable to a set of program labels ($\overline{\mathsf{Env}}_{\mathsf{PA}} : \mathsf{Var} \rightarrow \wp(\mathsf{Lab})$). We need a set of program labels as codomain since statically a variable could be related to references allocated at different program labels. Similarly, an abstract store relates a pair composed by a program label and a field name to a set of program labels ($\overline{\mathsf{Store}}_{\mathsf{PA}} : (\mathsf{Lab} \times \mathsf{Field}) \rightarrow \wp(\mathsf{Lab})$). Finally, abstract states are the Cartesian product of abstract environments and stores ($\overline{\Sigma}_{\mathsf{PA}} = \overline{\mathsf{Env}}_{\mathsf{PA}} \times \overline{\mathsf{Store}}_{\mathsf{PA}}$). The function $\overline{heapId}_{\mathsf{PA}} : \overline{\Sigma}_{\mathsf{PA}} \rightarrow \wp(\overline{\mathsf{HId}}_{\mathsf{PA}})$ returns the set of all the abstract memory locations in the environment and in the store. First of all, we

$$\begin{array}{l}\mathbb{E}_{\mathsf{PA}} : (\mathtt{rexp} \times \overline{\Sigma}_{\mathsf{PA}}) \rightarrow \wp(\mathsf{Lab}) \\ \mathbb{E}_{\mathsf{PA}}[\![\mathtt{x}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!] = \overline{\mathsf{e}}(\mathtt{x}) \\ \mathbb{E}_{\mathsf{PA}}[\![\mathtt{x.f}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!] = \bigcup_{\mathsf{l} \in \overline{\mathsf{e}}(\mathtt{x})} \overline{\mathsf{s}}(\mathsf{l}, \mathtt{f}) \\ \mathbb{E}_{\mathsf{PA}}[\![\mathtt{new}\ \mathtt{C}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!] = \{\mathit{label}(\mathtt{new}\ \mathtt{C})\}\end{array}$$

(a) The expression semantics of PA, where *label* given a statement returns its program label

$$\frac{\overline{\mathsf{s}}' = \bigsqcup_{\mathsf{l} \in \overline{\mathsf{e}}(\mathtt{x})} \overline{\mathsf{s}}[(\mathsf{l}, \mathtt{f}) \mapsto \overline{\mathsf{e}}(\mathtt{y})]}{\langle \mathtt{x.f} = \mathtt{y}, (\overline{\mathsf{e}}, \overline{\mathsf{s}}) \rangle \rightarrow^{\emptyset}_{\mathsf{PA}} (\overline{\mathsf{e}}, \overline{\mathsf{s}}')}$$

$$\frac{\overline{\mathsf{e}}' = \overline{\mathsf{e}}[\mathtt{x} \mapsto \mathbb{E}_{\mathsf{PA}}[\![\mathtt{rexp}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!]]}{\langle \mathtt{x} = \mathtt{rexp}, (\overline{\mathsf{e}}, \overline{\mathsf{s}}) \rangle \rightarrow^{\emptyset}_{\mathsf{PA}} (\overline{\mathsf{e}}', \overline{\mathsf{s}})}$$

$$\frac{}{\langle \mathtt{x.f}, (\overline{\mathsf{e}}, \overline{\mathsf{s}}) \rangle \rightarrow_{\mathsf{PA}} \bigcup_{\mathsf{l} \in \overline{\mathsf{e}}(\mathtt{x})} \overline{\mathsf{s}}(\mathsf{l}, \mathtt{f})}$$

(b) The statement semantics of PA

Fig. 13. PA definitions

collect all the labels that are actually stored in the abstract state. Formally, $\overline{\mathit{label}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) = \bigcup_{\mathtt{x} \in \mathit{dom}(\overline{\mathsf{e}}_{\mathsf{PA}})} \overline{\mathsf{e}}_{\mathsf{PA}}(\mathtt{x}) \bigcup_{(\mathsf{l}, \mathtt{f}) \in \mathit{dom}(\overline{\mathsf{s}}_{\mathsf{PA}})} \{\mathsf{l}\} \cup \overline{\mathsf{s}}_{\mathsf{PA}}(\mathsf{l}, \mathtt{f})$. Then $\overline{\mathit{heapId}}_{\mathsf{PA}}$ is defined as follows: $\overline{\mathit{heapId}}_{\mathsf{PA}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) = \bigcup_{\mathsf{l} \in \overline{\mathit{label}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}), \mathtt{f} \in \overline{\mathit{fieldVal}}_{\mathsf{PA}}(\mathsf{l})} (\mathsf{l}, \mathtt{f})$ where $\overline{\mathit{fieldVal}}_{\mathsf{PA}} : \mathsf{Lab} \rightarrow \mathsf{Field}$ is a function that, given a program point that contains the assignment of a `new` statement, returns all the possible value field names of the instantiated object.

The lattice structure relies on the pointwise application of set operators on $\overline{\mathsf{Env}}_{\mathsf{PA}}$ and $\overline{\mathsf{Store}}_{\mathsf{PA}}$, and the widening operator corresponds to the least upper bound operator since the set of program labels is finite. The concretization γ_{PA} first concretizes each label to a set of concrete references that could have been created at that program label, and then builds up the environments and stores in which abstract references are replaced by the corresponding references. Formally,

$$\begin{array}{l}\gamma_{\mathsf{PA}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) = \{((\mathsf{e}, \mathsf{s}), \gamma_{\overline{\mathsf{HId}}})\} : \\ \quad \gamma_{\overline{\mathsf{HId}}} \in \{[(\mathsf{l}, \mathtt{f}) \mapsto (\mathsf{r}, \mathtt{f}) : (\mathsf{l}, \mathtt{f}) \in \overline{\mathit{heapId}}_{\mathsf{PA}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) \wedge \mathsf{r} \in \mathit{allocatedRef}(\mathsf{l})]\} \\ \quad \mathsf{e} \in \{[\mathtt{x} \mapsto \mathsf{r} : \mathtt{x} \in \mathit{dom}(\overline{\mathsf{e}}) \wedge \mathsf{r} \in \bigcup_{\mathsf{l} \in \overline{\mathsf{e}}(\mathtt{x})} \gamma_{\overline{\mathsf{HId}}}(\mathit{allocatedRef}(\mathsf{l}))]\} \\ \quad \mathsf{s} \in \{[(\mathsf{r}_1, \mathtt{f}) \mapsto \mathsf{r}_2 : (\mathsf{l}_1, \mathtt{f}) \in \mathit{dom}(\overline{\mathsf{s}}) \wedge \mathsf{r}_1 \in \mathit{allocatedRef}(\mathsf{l}_1) \wedge \\ \qquad \mathsf{r}_2 \in \bigcup_{\mathsf{l}_2 \in \overline{\mathsf{s}}(\mathsf{l}_1, \mathtt{f})} \mathit{allocatedRef}(\mathsf{l}_2)]\}\end{array}$$

where $\mathit{allocatedRef} : \mathsf{Lab} \rightarrow \wp(\mathsf{Ref})$ is a function that returns all the references allocated by a given program label.

These definitions satisfy the soundness conditions of heap identifier concretization, and in particular **C1** since $\gamma_{\overline{\mathsf{HId}}}$ always concretizes all the heap identifiers in the state, **C2** since by definition $\sqcap_{\mathsf{PA}}$ is the pointwise application of the set intersection $\cap$, **C3** since what is allocated by a label is disjoint from what can be allocated by other labels, and **C4** since what is represented by a label never changes during the computation of the abstract semantics.

Semantics. Figure 13a formalizes the abstract evaluation of expressions, while Figure 13b deals with the semantics of statements. Both these semantics are quite standard. The evaluation of expressions simply enquires the environment or the store to know the abstract references pointed by a variable or a field access, respectively. Instead, when we create a new object, this returns a singleton containing the label of the statement. Similarly, the abstract semantics of statements assigns the set of labels returned by the evaluation of expressions to the assigned variable or abstract location. The semantics always creates an empty substitution, since statements do not change how we concretize the heap identifiers, because each heap identifier represents all the concrete locations allocated by a given label. This is not touched by the abstract semantics, and empty substitutions always satisfy Proposition 1. In addition, since only empty substitutions are produced, Proposition 2 trivially holds.

5.2 TVLA-Based Shape Analysis

Shape analysis [30] is an approach to heap analysis that achieved an impressive amount of research results, and it was used to define quite precise abstractions. TVLA [22] is the first and one of the most popular shape analysis engines. Ferrara *et al.* [15] combined TVLA and value analyses in a generic way relying on substitutions. A further work [16] has plugged this combination in the framework we introduced in this paper.

Intuitively, the concrete structure of the heap is represented by shapes defined by 2-valued logic structures. These are then approximated by 3-valued logic structures. At this level, *maybe* nodes represent summary node in the heap graph. Unfortunately, TVLA names nodes in a completely arbitrary and unpredictable way. Therefore, TVAL+ augmented states with unary name predicates. Condition **C3** imposes that different names point to different nodes. Therefore, each name predicate can point only to one node, and each node has to be pointed by one name predicate. The states satisfying this property are called *normalized*. When computing the TVLA semantics, the exit state may not be normalized. [15] then defines a normalization algorithm, and [16] proves that the substitutions it produces satisfy Propositions 1 and 2.

5.3 Numerical Domains

Numerical domains are by far the most studied value abstraction, and usually they track information on local variables. Our approach introduces heap identifiers in addition to variables. On the one hand, if a heap identifier represents a definite node (that is, it is always concretized to a single concrete reference by $\gamma_{\overline{\mathsf{HId}}}$), then the value domain can treat it exactly as a variable identifier. On the other hand, if it is a summary node (that is, it concretizes to many concrete references), the value analysis has to take into account this fact to preserve the soundness of the whole analysis.

There are three major issues in this scenario. First of all, when performing an assignment to a heap identifier representing a summary node, the value analysis has to perform a weak update, that is, it has to compute the least upper bound between the state before the assignment and after it. A similar issue arises when reading from a summary node. Imagine to analyze $\texttt{a} = \texttt{l.f}; \texttt{b} = \texttt{l.next.f};$ with the linear equalities domain [21] on the heap state depicted in the upper-left corner of Figure 3. The heap analysis would evaluate both `l` and `l.next` with `u1`. Therefore, after the computation of the semantics of $\texttt{a} = \texttt{u1.f}; \texttt{b} = \texttt{u1.f};$ we would infer that $\texttt{a} == \texttt{u1.f} \wedge \texttt{b} == \texttt{u1.f}$, and then that $\texttt{a} == \texttt{b}$, that is unsound. For this reason, when considering expressions containing summary nodes, the value domain has to consider that the same heap identifiers may represent different concrete heap locations. Finally, numerical domains usually deal with *all* the program variables. Instead, in our framework we cannot know a priori the heap identifiers produced by the heap approximation during the analysis. For instance, this would lead to situations in which we have to join value states defined on different environments.

All these issues are already well-known, and Gopan *et al.* [18] extended existing numerical domains (dealing only with local variables) to *summarized dimensions* in a generic way. In particular, they require four operators from a numerical domain (add, drop, fold, and expand). Using these operators, they define a sound semantics dealing with summary nodes. Their main insight is to (i) materialize one node from the summary node, (ii) perform the abstract evaluation or assignment on this node, and (iii) merge this node with the summary node where it comes from. If on the one hand this approach is quite precise, on the other hand it could introduce several identifiers when computing the semantics, slowing down the analysis.

6 Related Work

In this Section, we briefly discuss some previous work dealing with the combination of (usually shape) heap and (usually numerical) value analyses.

McCloskey et al. [25] proposed a generic way of combining heap and numerical domains. Similarly to our work, the heap analysis splits the heap into classes of disjoint portions of the heap as we did with heap identifiers (in particular with Condition **C3**). They assume that "the set of individuals belonging to a class is not affected by an assignment", that in our framework roughly means that what is represented by heap identifiers is not modified by assignments. Instead, one of the main focuses of our approach was to allow this scenario, and substitutions are the component used to communicate to the value analysis how heap identifiers are modified. In addition, they adopt first order logic formula to allow the analyses to communicate, and they require that the user of the analysis provides the predicates that are shared among the analyses. Instead, we automatically combined heap and value analysis, while we rely on reduce operators to communicate information from the value to the heap analysis. Similarly, Gulwani and Tiwari [19] rely on the Nelson-Oppen method to combine analyzers represented in first order logic, while our approach relies on abstract interpretation-based domains. Chang and Leino [6] relied on heap analyses based on equalities to allow the numerical domain to track information over heap locations. They extended the variable identifiers usually adopted by numerical domains with aliens expressions to track information over heap locations. Intuitively, this corresponds to our notion of heap identifiers.

There are only few previous works that combined generically heap and numerical domains based on abstract interpretation. Miné's memory abstraction [27], that is part of ASTRÉE [4], is parametrized on a numerical domain, but it does not support neither summary nodes, nor dynamic allocation. Abstract cofibered domains [32] (and in a more generic way the reduced cardinal power [10]) takes as argument a numerical domain, and they could be instantiated with various heap analyses. This framework requires to manually define the functor to glue the two domains, while our work is aimed at building a generic framework that *automatically* combines the two domains, and that relies on few assumptions to ensure the soundness of the whole analysis. Recently, Chang and Rival [7] introduced a modular combination of shape and numerical abstract domains. The shape analysis relies on points-to predicate, while the numerical domain tracks information on a symbolic representation of values stored in the heap. This is slightly different from our concept of heap identifiers, that are aimed at abstracting memory locations. In addition, this work targets shape analyses based on summarization and materialization of nodes. This implies that when a node is materialized, the shape analysis needs to track a disjunctive abstraction made by a set of shapes.

Several works dealt with refining the results of a specific heap analysis with some numerical information inferred by another analysis. In this context, Magill et al. [24] refine a heap analysis based on separation logic with some numerical domains through counter-examples generated by the shape analysis. Similarly, Beyer et al. [3] combined the model checker BLAST [2] with TVLA using Counter-Example Guided Abstraction Refinement for refining the shape analysis. Bouajjani et al. [5] developed a framework to statically infer properties over programs manipulating lists containing integer numerical data. Instead, our approach is generic both on the heap and value analysis, and the information tracked by the heap analysis could be refined by the value analysis through a reduce operator as described in Section 4.5.

7 Conclusion

In this paper we presented a sound generic framework to combine heap and value analyses automatically. Our framework relies on standard operators of static analyses based on abstract interpretation. In addition, it requires that the heap analysis provides a set of heap identifiers, how these identifiers are concretized into references, and some additional soundness conditions. As far as we know, this is the first generic combination that allows the heap analysis to merge and materialize heap identifiers. We instantiated our framework to a standard pointer and shape analyses as well as to numerical domains, thus proving empirically the expressiveness of our approach.

7.1 Future Work

The most part of the theoretical ideas contained in this paper came from some practical experience the authors get with `Sample` [8,14,17,34], a generic static analyzer that combines different heap and value analyses. We are currently extending this analyzer with all the results of this paper, and to provide an interface to plug implementation of heap and numerical analyses to external users.

Acknowledgments. This work was partially supported by the SNF project "Verification-Driven Inference of Contracts".

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT 9(5-6), 505–525 (2007)
3. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
4. 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: Proceedings of PLDI 2003. ACM (2003)
5. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
6. 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)
7. Chang, B.-Y.E., Rival, X.: Modular construction of shape-numeric analyzers. In: Festschrift for Dave Schmidt, EPTCS (2013)
8. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 505–521. Springer, Heidelberg (2011)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL 1977. ACM (1977)
10. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of POPL 1979. ACM (1979)
11. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13, 103–179 (1992)
12. Ferrara, P.: JAIL: Firewall analysis of java card by abstract interpretation. In: Proceedings of EAAI 2006 (2006)
13. Ferrara, P.: A fast and precise analysis for data race detection. In: Bytecode 2008 (2008)
14. Ferrara, P.: Static type analysis of pattern matching by abstract interpretation. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE 2010, Part II. LNCS, vol. 6117, pp. 186–200. Springer, Heidelberg (2010)

15. Ferrara, P., Fuchs, R., Juhasz, U.: TVAL+: TVLA and value analyses together. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 63–77. Springer, Heidelberg (2012)
16. Ferrara, P., Fuchs, R., Juhasz, U.: Tval+: A sound and generic combination of tvla and value analyses. Technical report, ETH Zurich (November 2013)
17. Ferrara, P., Müller, P.: Automatic inference of access permissions. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 202–218. Springer, Heidelberg (2012)
18. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 512–529. Springer, Heidelberg (2004)
19. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Proceedings of PLDI 2006. ACM (2006)
20. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of PASTE 2001. ACM (2001)
21. Karr, M.: On affine relationships among variables of a program. Acta Informatica 6(2), 133–151 (1976)
22. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)
23. 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)
24. 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)
25. 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)
26. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In: Proceedings of PLDI 2010. ACM (2010)
27. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: Proceedings of LCTES 2006. ACM (2006)
28. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation (2006)
29. Robert, V., Leroy, X.: A formally-verified alias analysis. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 11–26. Springer, Heidelberg (2012)
30. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
31. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 196–232. Springer, Heidelberg (2013)
32. 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)
33. Venet, A.: Towards the integration of symbolic and numerical static analysis. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 227–236. Springer, Heidelberg (2008)
34. Zanioli, M., Ferrara, P., Cortesi, A.: SAILS: static analysis of information leakage with Sample. In: Proceedings of SAC 2012. ACM (2012)

Modeling Parsimonious Putative Regulatory Networks: Complexity and Heuristic Approach

Vicente Acuña[1,2], Andrés Aravena[1,2,5], Alejandro Maass[1,2,3], and Anne Siegel[4,5]

[1] Center for Mathematical Modeling (UMI-CNRS 2807), University of Chile, Chile
[2] Center for Genome Regulation, University of Chile, Chile
[3] Department of Mathematical Engineering, University of Chile, Chile
[4] CNRS UMR 6074, IRISA Project Dyliss, Université de Rennes 1 (UMR 6074), France
[5] INRIA, Centre Rennes-Bretagne-Atlantique, Project Dyliss, Campus de Beaulieu, Rennes, France

Abstract. A relevant problem in systems biology is the description of the regulatory interactions between genes. It is observed that pairs of genes have significant correlation through several experimental conditions. The question is to find causal relationships that can explain this experimental evidence.

A putative regulatory network can be represented by an oriented weighted graph, where vertices represent genes, arcs represent predicted regulatory interactions and the arc weights represent the p-value of the prediction. Given such graph, and experimental evidence of correlation between pairs of vertices, we propose an abstraction and a method to enumerate all parsimonious subgraphs that assign causality relationships compatible with the experimental evidence.

When the problem is modeled as the minimization of a global weight function, we show that the enumeration of scenarios is a hard problem. As an heuristic, we model the problem as a set of independent minimization problems, each solvable in polynomial time, which can be combined to explore a relevant subset of the solution space. We present a logic-programming formalization of the model implemented using Answer Set Programming.

We show that, when the graph follows patterns that can be found in real organisms, our heuristic finds solutions that are good approximations to the full model. We encoded these approach using Answer Set Programming, applied this to a specific case in the organism *E. coli* and compared the execution time of each approach.

Keywords: Genic regulatory network reconstruction, Complexity, Algorithm, Heuristics.

Optimization methods and abstract methods such as static analysis or model-checking appear to have more and more interplays. For instance, optimization-based approaches are required in static analysis to handle the complexity of some numerical domains. The same issue holds in systems biology. In this domain, both static analysis and model-checking approaches have been widely developed (de Jong *et al.*, 2003; Calzone *et al.*,

2006; Chabrier-Rivier *et al.*, 2004; Danos *et al.*, 2007, 2012), but they are applied to models having strong litterature-based confidence and evidences. Meanwhile, the emergence of new sequencing technologies implies that more and more models of relatively low confidence are produced with learning and reconstruction approaches (Meyer *et al.*, 2007; Herrgård *et al.*, 2004; Marbach *et al.*, 2010). A main prospective issue is now to apply model-checking approaches to such uncertain models reconstructed from "omics" data.

As a first step in this direction, in this paper, we propose to use logic programming approaches to analyze "rough" data in order to build a robust and valid biological model that can be considered as a entry for formal approaches. More precisely, we address the issue of reconstructing a minimal graph model of regulatory interaction from genome and co-expression information. We prove that the underlying combinatorial problem is of high complexity, and we introduce a less complex (also NP-complete) heuristics to solve the problem. Then, interestingly, we propose to use a quite recent paradigm of logic programming, named Answer Set Programming (Gebser *et al.*, 2011), to solve the combinatorial problem. Interestingly, it appears that the progress of solvers developed for ASP now allows solving the heuristics that we have introduced.

1 Introduction

Molecular biology is source of many interesting graph problems. For instance, the transcriptional regulation network of an organism is usually represented by a directed graph where nodes represent genes and arcs connect each regulator gene to a regulated one. In theory the knowledge of the complete regulation network, including regulations signs (activation or repression), would allow a complete description of the cell behavior as a dynamical system (Xiao, 2009).

The set of genes is called the *genotype* of the organism, and the physical outcome that includes metabolites, proteins and the cell shape, constitute the *phenotype*. So the genotype is the potential outcome of a cell, while the phenotype is the effective outcome. The regulation network encodes the mechanism that enables a fixed genotype to become different phenotypes, for example when a multicellular organism develops and tissues are formed.

If we describe metaphorically a cell as a mechanical clock, the genetic information is the blueprint that describes each one of the gears. Genetic network reconstruction methods aim to describe how these gears are interconnected and how they interact for a given outcome. The long term goal is to describe accurately these interactions in a way that allow us to predict the effect of a change in the mechanism and, in principle, determine which modifications have to be made to obtain a desired result.

The regulation network can be modeled as a set of coupled differential equations, where each node is represented by a variable and the arcs represent the subset of these variables that are relevant for each equation. Unfortunately, these equations usually depend on parameters that are not easily measured. Another approach is to model them as boolean networks, where the nodes can be "active" or "inactive" and the next state of each node depends only on the state of the nodes connected by incoming arcs.

The purpose of gene regulation network reconstruction has thus at least two aspects. It constitutes new scientific knowledge as it describes the basis of the behavior of a cell. It is also the basis for biotechnological applications, like new antibiotics or genetic engineering (Davidson and Levin, 2005).

These networks are hard to determine experimentally (Streit *et al.*, 2013). Instead, many approximative methods build putative networks using pattern matching techniques in the DNA sequence (Bailey *et al.*, 2009). These methods have usually low specificity (Medina-Rivera *et al.*, 2011) and the resulting putative networks have a number of arcs around ten times larger than the expected. Putative networks are represented by weighted digraphs where each gene is represented by a vertex, an arc connects two vertices when the pattern matching suggests that the first gene regulates the second, and the arc weight is related to the pattern matching score or p-value.

Another approach to understand the genetic regulation is to observe the behavior of all genes through several conditions and determine correlations among genes that suggest that some of them are (indirectly) controlled by the same regulator. This is usually done using differential expression data from microarrays experiments. When the expression level of a gene is not independent from the level of another gene, they are called a pair of co-expressed genes. This association can be measured using linear correlation or mutual information, among other techniques (Butte and Kohane, 2000).

We can integrate these two kinds of data and use the experimental evidence as a constraint to define valid putative arc predictions and to determine parsimonious graphs that represent the transcriptional regulatory network. We are interested in the enumeration of all subgraphs that satisfy a connectivity restriction and are minimal in some sense. Our approach (Aravena *et al.*, 2013) to this parsimonious description is to find the subgraphs representing networks that have, for each pair of vertices representing co-regulated genes, at least one vertex that precedes them, directly or indirectly.

The arcs of the graph resulting from our method are novel targets for experimental validation. One of the advantages of our method over the classical tools is that the average degree is reduced, thus this validations can be focused on a few cases, decreasing experimental time and cost. Once some arc have been validated, this new knowledge can be easily incorporated into our model and close the loop between experiments and modeling. This iterative process alternating theoretical analysis and practical validation is classic in systems biology.

The paper is organized as follows: the next two sections explore parsimony by enumerating minimal subgraphs considering two definitions of graph minimality, the complexity of these problems is determined, in Section 4 an heuristic approach is described, finally in Section 5 this heuristic is applied to a well known organism.

2 Arc Minimal Subgraphs

In the following, $\mathcal{V}$ represents the set of all genes and $\mathcal{A}_0$ represents all putative regulatory relationships. We also have a collection $\mathcal{O} \subseteq \mathcal{P}_2(\mathcal{V})$ whose elements are subsets of $\mathcal{V}$ with cardinality 2, that is, unordered pairs $\{t, t'\}$ of distinct vertices (i.e. $t \neq t'$). This collection represents the pairs of co-regulated genes.

In order to obtain parsimonious regulatory graphs we need to compute subgraphs with a minimal set of arcs that can explain all experimental evidence. Thus, the solutions to our problem are completely defined by their set of arcs $A \subseteq \mathcal{A}_0$.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{A}_0)$ be a directed graph on vertex set $\mathcal{V}$ and arc set $\mathcal{A}_0$. A graph $G = (\mathcal{V}, A)$ is a **subgraph** of $\mathcal{G} = (\mathcal{V}, \mathcal{A}_0)$, if $A \subseteq \mathcal{A}_0$.

Now, we model the condition that for each pair of co-regulated genes our subgraph should contain a common regulator.

Definition 1. *Given an arc set $A \subseteq \mathcal{A}_0$ we say that a vertex $s \in \mathcal{V}$ **precedes** a vertex $t \in \mathcal{V}$ in A if there exists an oriented path from s to t using only arcs in A. In particular every node $v \in \mathcal{V}$ precedes itself.*

Definition 2. *We say that an arc set A is **$\mathcal{O}$-coherent** if each pair in $\mathcal{O}$ satisfies the **precedence condition**:*

$$\forall \{t, t'\} \in \mathcal{O} \quad \exists s \in \mathcal{V}, \quad s \text{ precedes } t \text{ in } A \wedge s \text{ precedes } t' \text{ in } A.$$

We also say that the subgraph $G = (\mathcal{V}, A)$ is $\mathcal{O}$-coherent when its arc set A is $\mathcal{O}$-coherent.

We assume that $\mathcal{A}_0$ is $\mathcal{O}$-coherent. Notice that, for each $\{t, t'\} \in \mathcal{O}$, if A contains a directed path from t to t' then the precedence condition is automatically satisfied by choosing $s = t$.

The idea is to describe the subsets of $\mathcal{A}_0$ which are $\mathcal{O}$-coherent. Notice that the property of being $\mathcal{O}$-coherent is monotone: if A is $\mathcal{O}$-coherent then every graph containing A is also $\mathcal{O}$-coherent. Thus, we are interested in enumerate only the subgraphs that are *minimal* in the following sense:

Definition 3. *We say that an $\mathcal{O}$-coherent arc set A is **minimal $\mathcal{O}$-coherent** if for any $a \in A$ we have that $A - a$ is not $\mathcal{O}$-coherent. We say that the subgraph $G = (\mathcal{V}, A)$ is minimal $\mathcal{O}$-coherent when its arc set A is minimal $\mathcal{O}$-coherent.*

Checking if a subgraph G is $\mathcal{O}$-coherent can be done in polynomial time. For each $\{t, t'\} \in \mathcal{O}$ we build the sets of all predecessors of t and all predecessors of t' in linear time. If the intersection is not empty for all pair $\{t, t'\} \in \mathcal{O}$ then G is $\mathcal{O}$-coherent. Therefore, it is easy to find *one* minimal $\mathcal{O}$-coherent subgraph of $\mathcal{G}$. By iteratively removing arcs of $\mathcal{G}$ while the condition is maintained we obtain a minimal graph in quadratic time. Consider the following problem:

> EnumCohe($\mathcal{G}, \mathcal{O}$): Given an oriented graph $\mathcal{G}$ and a set of pairs of vertices $\mathcal{O} \subset \mathcal{P}_2(V)$, enumerate all minimal $\mathcal{O}$-coherent subgraphs of $\mathcal{G}$.

We want to analyse the computational complexity of this enumeration problem. Notice that the number of minimal $\mathcal{O}$-coherent subgraphs of $\mathcal{G}$ can grow exponentially (consider, for instance, $\mathcal{A}_0$ a complete graph and $\mathcal{O}$ containing only one pair of vertices). Therefore, just printing the result would take exponential time in terms of the input size. In these cases, it is more appropriate to use *total time* to analyse the complexity of enumeration. That is, the time is measured in terms of the size of the input *and* the number of solutions (Johnson *et al.*, 1988). Thus, we say that EnumCohe can be done in *polynomial total time* if we can enumerate the solutions in polynomial time in the size of $\mathcal{G}$, $\mathcal{O}$ and the number of minimal $\mathcal{O}$-coherent subgraphs of $\mathcal{G}$.

Unfortunately, the problem EnumCohe is hard in following sense: enumerate all minimal $\mathcal{O}$-coherent subgraphs cannot be done in polynomial total time unless $P = NP$. To prove this, we reduce EnumCohe to the **path conjunction problem**:

> PathConj($\mathcal{G}, \mathcal{P}$): Given an oriented graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}_0)$ and a set of pairs of vertices $\mathcal{M} = \{(s_i, t_i), i = 1 \ldots n\} \subseteq \mathcal{V} \times \mathcal{V}$, enumerate all minimal subsets $A \subseteq \mathcal{A}_0$ such that for each $(s_i, t_i) \in \mathcal{M}$, there is an oriented path from s_i to t_i.

Here minimality is in the subset sense: if A is minimal then it connects all pairs in $\mathcal{M}$ and for each $a \in A$ there is at least one pair in $\mathcal{M}$ that is not connected in $A - a$. Khachiyan *et al.* (2007) shows that PathConj cannot be enumerated in polynomial total time unless $P = NP$.

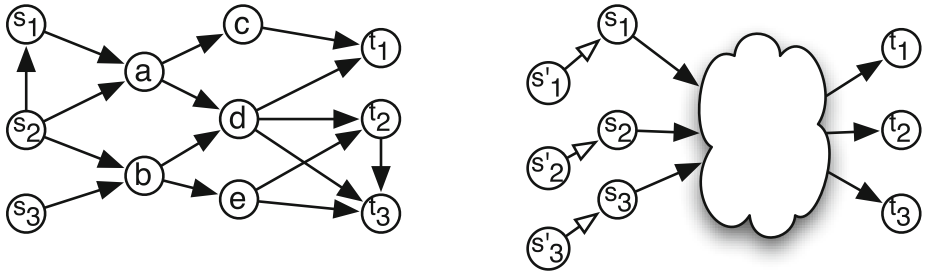


Fig. 1. (A) Example of the path conjugation problem, which enumerates all minimal subgraphs connecting pairs of vertices in $\mathcal{M} = \{(s_1, t_1), (s_2, t_2), (s_3, t_3)\}$. One such subgraph is the induced by the vertices a, b and d. **(B) Reduction of the path conjugation problem to EnumCohe**. Additions of the s'_i nodes guarantees that each s_i is connected to the corresponding t_i, as described in the text. The latter problem is thus as complex as the first.

Theorem 1. *Problem* EnumCohe *cannot be solved in polynomial total time unless $P = NP$.*

Proof. Problem PathConj can be reduced to EnumCohe in linear time. Let us consider $\mathcal{G} = (V, \mathcal{A}_0)$ and $\mathcal{M} = \{(s_i, t_i), i = 1 \ldots n\}$ an instance of PathConj.

We can create an instance for EnumCohe to solve this problem. Define the graph $G' = (V \cup V', \mathcal{A}_0 \cup \mathcal{A}_0')$ where $V' = \{s_i', i = 1 \ldots n\}$ and $\mathcal{A}_0' = \{(s_i', s_i), i = 1 \ldots n\}$. Consider the set of pairs $\mathcal{O} = \{(s_i', t_i), i = 1 \ldots n\}$. Clearly each minimal $\mathcal{O}$-coherent subgraph of G' is exactly the set of arcs in A' union a minimal subgraph connecting the pairs in $\mathcal{M}$. Then, there is a one-to-one correspondence between the solutions of EnumCohe$(G', \mathcal{O})$ and the solutions of PathConj$(\mathcal{G}, \mathcal{P})$.

3 Minimum Weight Subgraphs

The graphs that represent putative regulatory networks are built using pattern matching techniques that determine when a given gene can be a regulator (Altschul *et al.*, 1997) and which genes can it regulate (Bailey *et al.*, 2009) based on the DNA sequence of the genome. This prediction is characterized by the score of each gene versus the reference pattern, and by a p-value that states the probability of observing that score under the null hypothesis that there not exists a regulation relationship. A lower p-value corresponds to a higher confidence that the arc corresponds to a real regulatory relationships.

We assume that each arc in $\mathcal{A}_0$ has a positive weight that increases with the p-value of the arc. Then each subgraph has a global weight, and a parsimonious regulatory graph is any $\mathcal{O}$-coherent subgraph of minimum weight.

Let $w : \mathcal{A}_0 \to \mathbb{N}$ be the function that assigns a non-negative weight to each arc in $\mathcal{A}_0$. Then the weight (or cost) of an arc-set A is $W(A) = \sum_{a \in A} w(a)$. We are interested in finding a $\mathcal{O}$-coherent subgraph of minimum weight. It is easy to see that any minimum weight $\mathcal{O}$-coherent subgraph is also arc minimal, but not all arc minimal subsets have minimum weight. Unfortunately, even finding one $\mathcal{O}$-coherent subgraph of minimum weight is *NP*-hard. We define formally this problem as MinCohe:

> MinCohe$(\mathcal{G}, \mathcal{O})$: Given an oriented graph $\mathcal{G}$ and a set of pairs of vertices $\mathcal{O} \subset \mathcal{P}_2(\mathcal{V})$, find a $\mathcal{O}$-coherent subgraph of minimum weight.

To prove MinCohe is *NP*-hard, we introduce the Steiner Weighted Directed Tree problem:

> SWDT$(\mathcal{G}, s, T)$: Given an oriented weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}_0)$, a vertex $s \in \mathcal{V}$ and a set of vertices $T = \{t_i, i = 1 \ldots n\} \subseteq \mathcal{V}$, find a subgraph of minimum weight that connect s to t_i for all $t_i \in T$.

The problem SWDT is *NP*-hard. Indeed, the undirected case of this problem corresponds, in their decision version, to one of Karp's 21 *NP*-complete problems (Karp, 1972). Since SWDT is an extension of the undirected case, it is also *NP*-hard.

Theorem 2. *Problem* MinCohe *is NP-hard.*

Proof. We reduce SWDT problem to MinCohe in a similar way than in the previous result. Let us consider $\mathcal{G} = (\mathcal{V}, \mathcal{A}_0)$, $s \in V$ and $T = \{t_i, i = 1 \ldots n\}$ an

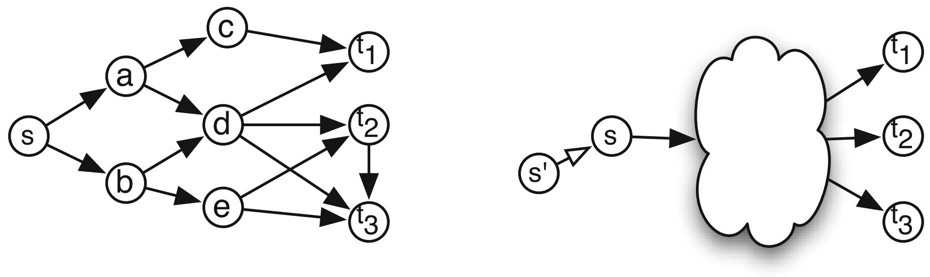


Fig. 2. (A) Schema of Steiner Directed Weighted Tree (SDWT), which enumerates all minimum weight subgraphs connecting s to vertices in $T = \{t_1, t_2, t_3\}$. For example the tree induced by nodes a and d connects s with T with minumum weight. **(B) Reduction of Steiner Directed Weighted Tree problem to MinCohe**. The latter problem is thus as complex as the first one.

instance of SWDT. Define the graph $G' = (\mathcal{V} \cup \{s'\}, \mathcal{A}_0 \cup \{(s', s)\})$ where s' is a new vertex and (s', s) is a new arc with weight zero. Consider the set of pairs $\mathcal{O} = \{(s', t_i), i = 1 \ldots n\}$. Clearly a solution of MinCohe$(G', \mathcal{O})$ is exactly the singleton $\{(s', s)\}$ union a solution of SWDT$(\mathcal{G}, s, T)$.

4 Subgraphs with Minimum Weight Paths

We define a *v-shape* as the union of two directed paths starting from the same vertex with no other vertex in common. Formally,

Definition 4. *Let s, t and t' be three vertices of G with $t \neq t'$. Let P be a directed path from s to t and let P' be a directed path from s to t' such that P and P' have only vertex s in common. Then, we say that $Q = P \cup P'$ is a* **v-shape**. *We also say that vertices t and t' are* **v-connected** *by Q.*

Clearly if an arc set $A \subseteq \mathcal{A}_0$ is $\mathcal{O}$-coherent, then for each pair $\{t, t'\}$ in $\mathcal{O}$ there is at least one v-shape in $G(\mathcal{V}, A)$ that v-connects t and t'. Thus, if we consider local parsimony principle, for each pair $\{t, t'\}$ in $\mathcal{O}$ we should include in our solution A a v-shape of minimum weight v-connecting t and t'.

This is not necessarily the case for the solutions given by MinCohe. Indeed, a solution G of MinCohe has minimum *global* weight, but this does not imply that every pair is v-connected by a minimum weight v-shape, as can be seen in Fig. 3.

In the following, we would like to consider only $\mathcal{O}$-coherent subgraphs that contain a minimum weight v-shape for each pair in $\mathcal{O}$. We first define the collection of all v-shapes of minimum weight connecting two vertices in our initial graph $G(\mathcal{V}, \mathcal{A}_0)$:

Definition 5. *Given a graph $G(\mathcal{V}, \mathcal{A}_0)$, we call* ***Short-v-shape***(t, t') *to the collection of all v-shapes that v-connect t and t' and are of minimum weight in $\mathcal{A}_0$.*

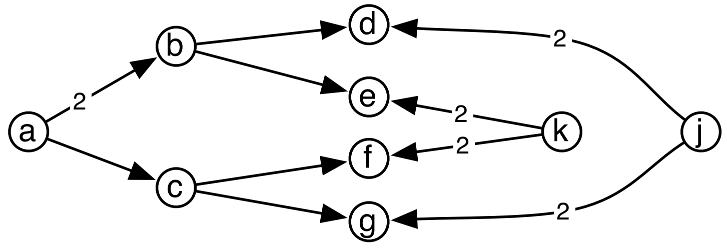


Fig. 3. Example graph where MinCohe solution is not formed by a minimum weight v-shapes. If $\mathcal{O} = \{\{d, g\}, \{e, f\}\}$ then the MinCohe solution has weight 7 and uses the arcs $(a, b), (b, d), (b, e), (a, c), (c, f), (c, g)$. An $\mathcal{O}$-short solution has weight 8. In contrast, when $\mathcal{O} = \{\{d, e\}, \{f, g\}\}$, both solutions coincide. Arcs have weight 1 unless otherwise declared.

Now, we can define the solutions that contain a minimum weight v-shape for every pair in $\mathcal{O}$.

Definition 6. *Given a $\mathcal{O}$-coherent arc set $A \subseteq \mathcal{A}_0$, we say that A is **$\mathcal{O}$-short** if the subgraph $G(\mathcal{V}, A)$ contains a v-shape in Short-v-shape(t, t') for each pair $\{t, t'\} \in \mathcal{O}$.*

We are interested in finding the $\mathcal{O}$-coherent subgraphs that are $\mathcal{O}$-short. In particular we are interested in those $\mathcal{O}$-short having minimum weight. We propose the following problem:

> MinWeightOshort$(\mathcal{G}, \mathcal{O})$: Given an oriented graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}_0)$ and a set of pairs of vertices $\mathcal{O} \subset \mathcal{P}_2(\mathcal{V})$, find a $\mathcal{O}$-short subgraph of minimum weight.

The following result is proved by a reduction from the NP-complete problem Hitting set (see Garey and Johnson, 1979): given a set of elements $A = \{1, \ldots, m\}$ and a collection of subsets $\mathcal{I} = \{I_1, \ldots, I_n\}$ of A, find a minimum cardinality subset of elements $H \subseteq A$ such that $H \bigcap I_i \neq \emptyset$, $\forall i = 1, \ldots, n$.

Theorem 3. *The problem* MinWeightOshort *is NP-hard.*

Proof. Let A and $\mathcal{I} = \{I_1, \ldots, I_n\}$ be an instance of hitting set problem. We consider the the graph $G(\mathcal{V}, A)$, where for each element a in A there are two vertices a and a' and an arc from a to a' of weight one. Additionally, for each set I_i with $i \in \{1, \ldots, n\}$ there are two vertices I_i and I'_i. Moreover, if a belongs to I_i , then there are two arcs of weight zero: one from vertex I_i to vertex a and one from vertex a' to vertex I'_i. If we define the set $\mathcal{O}$ by including all the pairs of vertices $\{I_i, I'_i\}$, then clearly any $\mathcal{O}$-short subgraph of minimum weight correspond to a minimum cardinality hitting set of the original problem.

Although this problem is theoretically hard, it could be much more tractable than the previous formulations for the instances that we are interested. Indeed, the combinatorial explosion of feasible solutions can be controlled if the size of the collections *Short-v-shape*(t, t') is small for every pair $\{t, t'\}$ in $\mathcal{O}$. That is, the number of v-shapes of minimum weight between each pair of vertices in $\mathcal{O}$ is small.

Thus, we can use a complete enumeration of unions generated by choosing one v-shape for each pair. At the end we select those unions of minimum weight.

Notice that, for a pair $\{t, t'\} \in \mathcal{O}$, computing the set *Short-v-shape*(t, t') can be done in polynomial total time by using some clever modification of the Dijkstra's algorithm (Dijkstra, 1959).

5 An Illustrative Example

To evaluate these approaches we consider a toy problem on a well known organism. We build a graph $\mathcal{G}$ using the genomic DNA sequence of the bacteria *E. coli* and patterns described in RegulonDB. We identified as putative regulators the genes with high homology to know genes coding for transcription factors following a standard protocol: using Blast (Altschul *et al.*, 1997) with a cutoff E-value of 10^{-10}. Then we determined where these transcription factors could bind using the tool FIMO from the MEME suite (Bailey *et al.*, 2009) to determine the presence of putative binding sites in the upstream region of every gene, with a p-value cutoff of 10^{-5}. When these binding sites were found we connected with an arc each regulator gene to the gene located downstream the binding site. This protocol is basically the classical protocol used for regulatory network reconstruction.

As described, the arcs of this graph are inferred using probabilistic tools that characterize them with a E-value and a p-value. We combined these two values and ranked all the arcs into three categories: high, medium and low confidence. We assigned discrete weights to each arc according to this classification. High confidence arcs have weight equal to 1, medium confidence arcs have weight 10 and high confidence arcs have weight 100.

We determined the set $\mathcal{O}$ of pairs of co-expressed genes from a set of 133 differential expression experiments, estimating the mutual information among them using the Pearson method and choosing the relevant relationships by the *Maximum Relevance Minimum Redundancy* (MRNET) criteria (Meyer *et al.*, 2007). This method uses mutual information as a way to determine non-linear dependencies between the expression profiles of the genes, and then determines which dependencies are significant with the following iterative procedure: for each gene a, it determines a set S_a of potentially associated genes. Initially $S_a = \varnothing$. In each iteration MRNET determines the gene b that maximizes

$$MI(a, b) - \frac{1}{|S_a|} \sum_{c \in S_a} MI(b, c)$$

The gene b that maximizes this expression with a value over a threshold is added to the set S. This expression corresponds perfectly to the idea behind MRNET. The first term of this expression focus on finding the associated genes that are of maximal relevance for a, while the second term focus on minimizing the redundancy with respect to the associated genes already in S_a. Under these conditions the pair (a, b) is in $\mathcal{O}$. We further limited $\mathcal{O}$ to the 10 000 elements with higher mutual information.

Finally, to include an aditional biological constraint and reduce the network size we contracted the graph $\mathcal{G}$ and the set $\mathcal{O}$ using the node equivalence classes defined by *operons* as predicted in ProOpDB (Taboada *et al.*, 2012). This can be done because, in bacteria and other prokaryotes, operons are set of genes that are always expressed together so their behavior is identical.

The graph $\mathcal{G}$ contains 2215 vertices and 11 584 arcs, the set $\mathcal{O}$ contains 9442 pairs of vertices. We relied on Boolean constraint processing technology for coding $\textsc{MinCohe}(\mathcal{G}, \mathcal{O})$, given that it is highly effective for solving demanding combinatorial problems. To be more precise, we take advantage of Answer Set Programming (ASP), a declarative problem solving approach providing a declarative framework for modeling various Knowledge Representation and Reasoning problems (Baral, 2003) combining a rich yet simple modeling language with high-performance Boolean constraint solving capacities.

The pairing of declarativeness and performance in state-of-the-art ASP solvers allows for concentrating on an actual problem, rather than how to implementing it. The basic idea of ASP is to express a problem in a logical format so that the models of its representation provide the solutions to the original problem. Problems are expressed as logic programs and the resulting models are referred to as answer sets. Although determining whether a program has a answer set is the fundamental decision problem in ASP, more reasoning modes are needed for covering the variety of reasoning problems encountered in applications. Hence, a modern ASP solver, like clasp supports several reasoning modes for assessing the multitude of answer sets, among them, regular and projective enumeration, intersection and union, and multi-criteria optimization. As well, these reasoning modes can be combined, for instance, for computing the intersection of all optimal models. This is accomplished in several steps. At first, a logic program with first-order variables is turned by efficient database techniques into a propositional logic program. This is in turn passed to a solver computing the answer sets of the resulting program by using advanced Boolean constraint technology. For optimization, a solver like clasp uses usually branch-and-bound algorithms (other choices, like computing unsatisfiable cores, exist). The enumeration of all optimal models is done in two steps. At first an optimal model is determined along with its optimum value. This computation has itself two distinct phases. First, an optimal model candidate must be found and second, it must be shown that there is no better candidate; the latter amounts to a proof of unsatisfiability and is often complex. Then, all models possessing the same value are enumerated in a second step.

Our encodings are written in the input language of gringo 3 (Gebser *et al.*, 2009, 2011). In what follows we introduce its basic syntax and we refer the reader to the corresponding literature for more details. An atom is a constant (e.g. `p`, `q`) or a function symbol (e.g. `p(a,b)`, `q(X,10)`) where uppercase denotes first-order variables. Then, a rule is of the form

$$H\texttt{:-}\ B_1, \ldots, B_n.$$

where H (head) is an atom and any B_j (body) is a literal of the form A or `not` A for an atom A where the connective `not` corresponds to default negation. Further, a rule without body is a fact, whereas a rule without head is an integrity constraint. A logic program consists of a set of rules, each of which is terminated by a period. The connectives `:-` and `,` can be read as *if* and *and*, respectively. A statement starting with `not` is satisfied unless its enclosed proposition is found to be true. The semantics of a logic program is given by the *stable models semantics* (Gelfond and Lifschitz, 1988). Intuitively, the head of a rule has to be true whenever all its body literals are true. In ASP every atom needs some derivation, i.e., an atom cannot be true if there is no rule deriving it. This implies that only atoms appearing in some head can appear in answer sets, i.e. stable models. We end this quick introduction by three language constructs particularly interesting for our encoding. First, the so called choice rule of the form,

$$\{H_1, \ldots, H_m\}\texttt{:-}\ B_1, \ldots, Bn.$$

allows us to express choices over subsets of atoms. Any subset of its head atoms can be included in a stable model, provided the body literals are satisfied. Note that using a choice rule one can easily generate an exponential search space of candidate solutions. Second, a conditional literal is of the form

$$L : L1 : \cdots : Ln$$

The purpose of this language construct is to govern the instantiation of the literal L through the literals $L_1, \ldots, L_n$. In this respect, the conditional literal above can be regarded as the list of elements in the set $\{L|L_1, \ldots, L_n\}$. Finally, for solving (multi-criteria) optimization problems, ASP allows for expressing cost functions in terms of a weighted sum of elements subject to minimization and/or maximization. Such objective functions are expressed in ASP in terms of optimization statements of the form

$$\texttt{\#minimize}[L_1 = W_1@P_1, \ldots, L_N = W_N@P_N].$$

where every L_j is a literal and every W_j an integer weight. Further, P_i provides an integer priority level. Priorities allow for representing lexicographically ordered minimization objectives, greater levels being more significant than smaller ones. By default all priorities are 1.

The coding, shown in Fig 4, is straight-forward. Predicates `arc(X,Y,W)` represent the arcs in $\mathcal{A}_0$ and their weights, and predicates `coexp(X,Y)` represent the elements of $\mathcal{O}$. The optimization is carried on in two stages. First the solver looks for the minimum possible global weight. Then, once this value has been determined, we look for all the answer sets that realize the minimum values. In each answer set the predicates `used_arc(X,Y,W)` indicate the arcs of a subgraph satisfying MinCohe($\mathcal{G}, \mathcal{O}$).

Execution of this program is highly time-consuming. After a week of clock time we reached the time limit of our cluster scheduler without finding the minimum weight value.

```
% Input: arc(X,Y,W) means there is an arc between X and Y with weight W
% Input: coexp(X,Y) means that {X,Y} are in O

% each arc can be used or not
{ used_arc(X,Y,W) } :- arc(X,Y,W).

% node X precedes node Y
precedes(X,Y) :- used_arc(X,Y,_).
precedes(X,Y) :- precedes(X,Z), used_arc(Z,Y,_).

% motif M is an explanation of operons A and B linked by coexpressedOp/2
v_connected(A,B) :- precedes(M,A), precedes(M,B), coexp(A,B).

% all coexpressed vertices should be v-connected
:- coexp(A,B), not v_connected(A,B).

% look for minimum global weight
#minimize [used_arc(X,Y,W)=W].
```

Fig. 4. ASP code to find a solution of MinCohe

```
% Input: vshape(I,A,B) when v-shape I is in short-v-shapes(A,B)
% Input: arcInVshape(I,X,Y,W) when v-shape I has an arc (X, Y) w/weight W
% Input: coexp(X,Y) means that {X,Y} are in the set O

% only one v-shape is chosen for each {t,t'} in O
1{ chosen(I) : vshape(I,A,B) }1 :- coexp(A,B).

% consider the arcs that are part of the chosen v-shape
used_arc(X,Y,W) :- arcInVshape(I,X,Y,W), chosen(I).

% minimize the global weight
#minimize [used_arc(_,_,W) = W].
#hide.
#show used_arc/3.
```

Fig. 5. ASP code to find a solution of MinCohe

We then proceeded to solve MinWeightOshort($\mathcal{G},\mathcal{O}$) using the following strategy. For each $\{t,t'\} \in \mathcal{O}$ we determine the set *Short-v-shape*(t,t') using the `get.all.shortest.paths` of the *igraph* library (Csardi and Nepusz, 2006) in the R environment (R Core Team, 2012), and assigned an unique id to each one. We coded these v-shapes using the ASP predicate `vshape(ID,T1,T2)` and the arcs that form them with the predicate `arcInVshape(ID,X,Y,W)`. In this encoding `ID` corresponds to the v-shape id, `T1,T2` correspond to $t,t' \in \mathcal{O}$, `X,Y` identify the extremes of an arc, and `W` is the weight of it.

Using these predicates, and the rules in Fig. 5, we used ASP solver *unclasp* to find the minimum weight. Execution time was 15 seconds.

A second execution was performed to find all answer sets realizing that weight. Notice that this encoding can describe the same graph as combinations of different v-shapes. We used the meta-commands `#hide`, `#show used_arc/3` and the *clasp* option `project` to collapse all answer sets with the same `used_arc/3` predicates in a single answer. This second execution took 80 minutes and resulted in a unique graph.

6 Conclusion

The proposed algorithm can enumerate MinWeightOshort solutions in practical time, providing a way to explore a relevant subset of the $\mathcal{O}$-coherent subgraphs significantly faster than solving MinCohe. In many cases, when the graph represents a real regulatory network, it is reasonable to expect that many co-expressed nodes are connected by short v-shapes. In such cases the proposed algorithm can be used as an heuristic for MinCohe.

When it is relevant to find an exact solution of MinCohe, the heuristic solution is still useful. First, it provides a good upper bound for the global weight, which can speed up the search for the optimal value. Second, a solution of MinWeightOshort is a graph that can be used as a starting point for the combinatorial exploration required by MinCohe. We think this can be applied using the new heuristic ASP solver *hclasp* in the Potassco suite.

Acknowledgements. We acknowledge the support by Fondap [15090007]; Basal Grant CMM; IntegrativeBioChile INRIA Associated Team; CIRIC-INRIA Chile line Natural Ressources; ANR Biotempo[ANR-10-BLANC-0218]; U. Européenne de Bretagne [to A.A.]; INRIA-Conicyt 2010–55 [to A.A. and A.M.]; and "Estadías de Investigación U. de Chile" [to A.A.].

References

Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped blast and psi-blast: a new generation of protein database search programs. Nucleic Acids Res. 25(17), 3389–3402 (1997)

Aravena, A., Guziolowski, C., Ostrowski, M., Schaub, T., Eveillard, D., Maass, A., Siegel, A.: Deciphering regulatory relationships with a logic-based model of causality for gene expression associations (2013) (in preparation)

Bailey, T.L., Boden, M., Buske, F.A., Frith, M., Grant, C.E., Clementi, L., Ren, J., Li, W.W., Noble, W.S.: Meme suite: tools for motif discovery and searching. Nucleic Acids Research 37 (Web Server issue), W202 (2009)

Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)

Butte, A.J., Kohane, I.S.: Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In: Pac. Symp. Biocomput., pp. 418–429 (2000)

Calzone, L., Fages, F., Soliman, S.: Biocham: an environment for modeling biological systems and formalizing experimental knowledge. Bioinformatics (2006)

Chabrier-Rivier, N., Chiaverini, M., Danos, V., Fages, F., Schächter, V.: Modeling and querying biomolecular interaction networks. Theoretical Computer Science 325(1), 25–44 (2004)

Csardi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal, Complex Systems 1695 (2006)

Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-based modelling of cellular signalling. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 17–41. Springer, Heidelberg (2007)

Danos, V., Feret, J., Fontana, W., Harmer, R., Hayman, J., Krivine, J., Thompson-Walsh, C.D., Winskel, G.: Graphs, rewriting and pathway reconstruction for rule-based models. In: D'Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012. LIPIcs, vol. 18, pp. 276–288. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012)

Davidson, E., Levin, M.: Gene regulatory networks. Proceedings of the National Academy of Sciences of the United States of America 102(14), 4935 (2005)

de Jong, H., Geiselmann, J., Hernandez, C., Page, M.: Genetic network analyzer: qualitative simulation of genetic regulatory networks. Bioinformatics 19(3), 336–344 (2003)

Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik 1(1), 269–271 (1959)

Garey, M.R., Johnson, D.S.: Computers and Intractability (A guide to the theory of NP-completeness). W.H. Freeman and Company, New York (1979)

Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder *gringo*. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 502–508. Springer, Heidelberg (2009)

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The potsdam answer set solving collection. AI Communications 24(2), 107–124 (2011)

Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, vol. 88, pp. 1070–1080 (1988)

Herrgård, M.J., Covert, M.W., Palsson, B.Ø.: Reconstruction of microbial transcriptional regulatory networks. Curr. Opin. Biotechnol. 15(1), 70–77 (2004)

Johnson, D., Yannakakis, M., Papadimitriou, C.: On generating all maximal independent sets. Information Processing Letters 27(3), 119–123 (1988)

Karp, R.M.: Reducibility Among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)

Khachiyan, L., Boros, E., Elbassioni, K., Gurvich, V., Makino, K.: Enumerating disjunctions and conjunctions of paths and cuts in reliability theory. Discrete Applied Mathematics 155(2), 137–149 (2007)

Marbach, D., Prill, R.J., Schaffter, T., Mattiussi, C., Floreano, D., Stolovitzky, G.: Revealing strengths and weaknesses of methods for gene network inference. In: Proceedings of the National Academy of Sciences (2010)

Medina-Rivera, A., Abreu-Goodger, C., Thomas-Chollier, M., Salgado, H., Collado-Vides, J., van Helden, J.: Theoretical and empirical quality assessment of transcription factor-binding motifs. Nucleic Acids Research 39(3), 808–824 (2011)

Meyer, P.E., Kontos, K., Lafitte, F., Bontempi, G.: Information-theoretic inference of large transcriptional regulatory networks. EURASIP J. Bioinform. Syst. Biol., 79879 (2007)

R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2012). ISBN 3-900051-07-0

Streit, A., Tambalo, M., Chen, J., Grocott, T., Anwar, M., Sosinsky, A., Stern, C.D.: Experimental approaches for gene regulatory network construction: The chick as a model system. Genesis 51(5), 296–310 (2013)

Taboada, B., Ciria, R., Martinez-Guerrero, C.E., Merino, E.: Proopdb: Prokaryotic operon database. Nucleic Acids Res. 40(Database issue), D627–D631 (2012)

Xiao, Y.: A tutorial on analysis and simulation of boolean gene regulatory network models. Current Genomics 10(7), 511 (2009)

Practical Floating-Point Tests with Integer Code

Anthony Romano

Stanford University

Abstract. Testing integer software with symbolic execution is well-established but floating-point remains a specialty feature. Modern symbolic floating-point tactics include concretization, lexical analysis, floating-point solvers, and intricate theories, but mostly ignore the default integer-only capabilities. If a symbolic executor is already high-performance, then software-emulation, common to integer-only machines, becomes a compelling choice for symbolic floating-point.

We propose a software floating-point emulation extension for symbolic execution of binary programs. First, supporting a soft floating-point library requires little effort, so multiple models are cheap; our executor has five distinct open source soft floating-point code bases. For integrity, test cases from symbolic execution of library code itself are hardware validated; mismatches with hardware appear in every tested library, a just-in-time compiler, a machine decoder, and several floating-point solvers. In practice, the executor finds program faults involving floating-point in hundreds of Linux binaries.

1 Introduction

Symbolic execution [29] is a popular dynamic analysis technique [40] for finding execution paths leading to program errors. A binary symbolic executor [7,9,18,23,35] explores paths by interpreting a program binary with symbolic expressions and forking the program state on contingent conditions. A state accrues path constraints (its model) by following feasible conditional branches. To obtain a satisfying variable assignment for a path (a test case), constraints are usually cast over a theory of bit-vectors and arrays which is solved with a decision procedure or theorem prover. Deciding satisfiability modulo the theory of bit-vectors is meant for integer workloads; expressions are built from two's complement arithmetic, integer comparisons, and bitwise operators.

Solving for expressions over floating-point operations requires additional effort and is considered a significant challenge in model checking [1]. There is little agreement on how to best handle symbolic floating-point data in a symbolic executor; in fact, several classes of floating-point support have been proposed. The simplest support evaluates only concrete data [8], which is fast and sound, but incomplete. Another approach, but still incomplete, applies taint analysis [17] and floating-point expression matching [11] to detect suspicious paths. The most challenging, complete and accurate symbolic floating-point semantics, relies on the flawless reasoning of a floating-point solver [2,3,5].

Accurate floating-point is essential for checking software. The de facto floating-point standard, IEEE-754 [25], fully describes a floating-point arithmetic model; it is subtle and complicated. A cautious software author must account for lurid details [20] such as infinities, not-a-numbers (`NaN`s), denormals, and rounding modes. Ignoring these details is convenient once the software appears to work but defects then arise from malicious or unintended inputs as a consequence.

Fortunately, IEEE-754 emulation libraries already encode the specifics of floating-point in software. Namely, soft floating-point emulates IEEE-754 operations with integer instructions. These libraries are a fixture in operating systems; unimplemented floating-point instructions trap into software handlers. Elsewhere, soft floating-point shared libraries assist when the instruction set lacks floating-point instructions.

This work presents an integer-only binary symbolic executor augmented to support floating-point instructions through soft floating-point libraries. Five off-the-shelf soft floating-point libraries are adapted to the symbolic executor by mapping soft floating-point library operations to a unified runtime interface. Floating-point instructions are mapped to integer code by replacing program instructions with calls to soft floating-point runtime functions. Aside from testing floating-point paths in general purpose programs, binary symbolic execution with soft floating-point provides a novel means for testing floating-point semantics in floating-point emulation libraries and floating-point theories.

The rest of this paper is structured as follows. Section 2 discusses related work, relevant background, and the motivation behind soft floating-point in a symbolic executor. Section 3 describes the operation and implementation of soft floating-point in a binary symbolic executor. Section 4 analyzes operation integrity through symbolic execution of soft floating-point libraries with symbolic operands and comparing the generated test cases against native evaluation on host hardware. Section 5 continues further by testing floating-point SMT solvers for inconsistencies with hardware. Section 6 considers general purpose applications by examining errors in Linux binaries found through symbolic execution with soft floating-point. Finally, Section 7 concludes.

2 Related Work

Testing and verification of floating-point software is the topic of much study. At the most primitive level, finite-width floating-point variables are reconciled with real and rational arithmetic and are the focus of floating-point decision procedures. Static analysis of source code leverages these primitives to find floating-point bugs but is limited to only an approximation of execution. A symbolic executor of floating-point code dynamically executes programs and must balance performance, completeness, and soundness with different workload needs. If floating-point data is concrete or uninteresting symbolically, symbolic execution of floating-point operations may be exclusively concrete. Fast, but unsound, symbolic floating-point, useful for bug finding, applies canonicalization and matching on floating-point expressions. When complete semantics are necessary, precise symbolic floating-point integrates a floating-point solver into the symbolic execution stack. This work introduces a new point in this progression: symbolically executed *soft* floating-point with integer code.

Many techniques have been developed to handle abstract floating-point data. Abstract domains [13], interval propagation [14], and abstraction refinement [10] are some influential approaches for computing solutions to value constraints. Such concepts have been extended, improved, and refined for floating-point values through exact projections [5], filtering by maximum units in the last place [2], interval linear forms [34], monitor variables [26], saturation with simplex bounding [12], conflict analysis over lattice abstractions [22], and guided approximation transformations of formulas [6], among others. For ease of use, decision procedures based on these strategies may be integrated into a solver back-end [2,5,6,12,22]. For hardware, formal floating-point specifications have been used to verify the correctness of a gate-level description of a floating-point unit [38].

Static analysis of source code to find floating-point bugs includes a broad class of notable abstract interpretation systems. The FLUCTUAT [21] system models floating-point values in C programs with affine arithmetic over noise symbols to locate sources of rounding error with respect to real numbers. ASTRÉE [4] is based on an interval abstraction that uses a multi-abstraction, specializable, domain-aware analysis to prove the absence of overflows and other errors for source programs written in a subset of the C language.

The simplest floating-point symbolic execution tactic discards symbolic data in favor of processing floating-point operations through concretization [8]. To dispatch a floating-point operation on symbolic data, each operand is constrained to a satisfying variable assignment (concretized) and the operation is evaluated. As an example, if x is unconstrained and 0 is the default assignment, computing $x + 1.0$ concretizes to $0.0 + 1.0$. Concretization is fast but it overconstrains the variable term x and discards most feasible values (i.e., $x \neq 0.0$).

A symbolic executor with expression matching also avoids the difficulty of supporting full floating-point semantics. These symbolic executors broaden the expression language to include floating-point operators but only consider the structure of the expressions. Taint tracking is one instance of expression analysis on symbolic floating-point; floating-point operations tag expressions and dereferences of tagged pointers are flagged [17]. Beyond tainting, comparison of expression structure [11,32] demonstrates equivalence between algorithm implementations. This analysis is often unsound; spurious errors are possible.

Precise symbolic floating-point reasons about the underlying semantics with the assistance of a floating-point solver. Although explicit reasoning is accurate and often complete by design, it demands a careful solver implementation (tested in Section 5) and invasive executor modifications [3]. In some cases, authors of these systems use the symbolic executor as a platform to test their own floating-point decision algorithms in lieu of a third-party IEEE-754 solver [2,5,31].

This paper proposes symbolically executed soft floating-point, a compromise between concretization and full symbolic floating-point. Where a floating-point solver may model a formula as disjunctions of several feasible subformulas, soft floating-point models that formula with many states. A soft floating-point operation partially concretizes on program control at contingent branches by forking into multiple feasible states. These states partition floating-point values by disjunction but together represent the set of all feasible floating-point values. Additional states are costly but soft floating-point is still attractive because the complicated floating-point component is off-the-shelf software which requires little support code and no changes to the core symbolic execution system. Aside from its simplicity, soft floating-point is self-testing, a property explored in Section 4 and applied to floating-point solvers in Section 5.

3 Soft Floating-Point

This section details the implementation of a soft floating-point extension for an integer-only symbolic binary executor. First, an abstract soft floating-point library is defined by its set of floating-point operations. Next, the components for the base binary symbolic executor are briefly outlined. Finally, a description of a runtime interface and implementation establishes the connection between the symbolic executor and several soft floating-point libraries.

3.1 Floating-Point Operations

A soft floating-point library is a collection of idempotent integer-only operation functions which model an IEEE-754-1985 [25] compliant floating-point unit. The client code bitcasts floating-point data (single or double precision) into integer operands; the library unpacks the sign, mantissa, and exponent components (Figure 1) with bitwise operators into distinct integers. Operations evaluate the components, then pack and return floating-point results in IEEE-754 format.

	Sign	**Exponent**	**Mantissa**	
31	1 bit	8 bits	23 bits	0

Single Precision

	Sign	**Exponent**	**Mantissa**	
63	1 bit	11 bits	52 bits	0

Double Precision

Fig. 1. IEEE-754 format for single and double precision floating-point data

Arithmetic. IEEE-754 defines the four-function arithmetic operations and remainder: $+$, $-$, $*$, $/$, and $\%$. Arithmetic operations are complete floating-point valued functions over single and double-precision pairs. A major repercussion of floating-point arithmetic is many desirable invariants from real numbers and two's-complement are lost: addition is non-associative, subtraction has cancellation error, and division by zero is well-defined.

Comparisons. Conditions on floating-point values are computed with comparison functions. Comparisons are defined for all pairs of 32-bit and 64-bit floating-point values and are represented with the usual symbols (i.e., $=$, $\neq$, $>$, $<$, $\geq$, and $\leq$). Evaluation returns the integer 1 when true, and 0 when false.

Comparisons take either an ordered or unordered mode. The mode determines the behavior of the comparison on non-number values. An ordered comparison may only be true when neither operand is a `NaN`. An unordered comparison is true if either operand is a `NaN`. During testing, only ordered comparisons were observed in code, so the two were never confused.

Type-Conversion. Type conversion translates a value from one type to another; floating-point values may be rounded to integers and back, or between single and double precision. In general, rounding is necessary for type conversion. Additionally, values may be rounded to zero, down to $-\infty$, up to ∞, or to nearest, depending on the rounding mode. However, only the round nearest mode appeared in program code during testing. There are several ways a floating-point computation may be rounded for type conversion:

- **Truncation and Expansion** ($\leftrightarrow$). Data is translated between single and double precision. Mantissa bits may be lost and values can overflow.
- **Integer Source** ($f \leftarrow i$). Conversion from integer to float. The integer may exceed the mantissa precision.
- **Integer Target** ($f \rightarrow i$). Conversion from float to integer; `NaN` and ∞ values may be converted.

Elementary Functions. The only elementary function required by IEEE-754-1985 is the square root function. Hence, all soft floating-point libraries support it. Transcendental functions, on the other hand, were deemed too costly to due to the precision necessary for correctness to the half-unit in the last place (the table-maker's dilemma [27]). Although present in some instruction sets (e.g., x86), transcendental functions are treated as unsupported non-standard extensions.

3.2 Binary Symbolic Executor

The binary symbolic executor uses the KLEE [8] LLVM symbolic interpreter, the STP [16] SMT solver, and the VEX [37] machine instruction decoder. A modified KLEE interpreter dispatches instructions by decoding program binaries from machine code into LLVM with a custom VEX-to-LLVM dynamic binary translator (DBT). In addition, the system has a just-in-time DBT (the JIT) built on the VEX-to-LLVM translator and the LLVM JIT engine; in Section 4 the JIT tests the interpreter's fidelity to bitcode semantics on concrete data.

3.3 Runtime Libraries

The soft floating-point extended symbolic interpreter handles floating-point operations by calling out to a runtime library with a standard interface. All floating-point instructions are replaced with runtime function calls that manipulate

Table 1. Rewritten LLVM instructions with corresponding SoftFloat function calls

LLVM Instruction	SoftFloat Function
`FAdd`, `FSub`	float{32,64}_{add,sub}
`FMul`, `FDiv`, `FRem`	float{32,64}_{mul,div,rem}
`FCmp`	float{32,64}_{lt,le,eq}
`FPExt`	float32_float64
`FPTrunc`	float64_float32
`FPToSI`	float{32,64}_int{32,64}
`SIToFP`	int{32,64}_float{64,32}
`sqrtf`, `sqrt`	float{32,64}_sqrt

floating-point data with integer instructions. Internally, the runtime libraries differ on a fundamental level by the data encoding used for computation. For porting a library to the interpreter, floating-point emulation code from open-source operating systems supplies the majority of the library implementations.

A soft floating-point library, loaded as part of the KLEE LLVM bitcode runtime, encodes data operations in integer terms with an idempotent function call interface. If the library is correct, then every floating-point operation is completely modeled; there is no need to re-encode the details. To map floating-point code to the integer-only interpreter, the program code is rewritten with soft floating-point calls. To validate the design, the runtime supports five off-the-shelf soft floating-point implementations: `bsdhppa` (PA-RISC from NetBSD), `bsdppc` (PowerPC from NetBSD), `linmips` (MIPS from Linux), `softfloat` (the SoftFloat library [24]), and `shotgun` (from an emulator [28]).

Instruction Rewriting. A function pass rewrites program code to call soft floating-point runtime functions in place of LLVM floating-point instructions. The pass replaces every floating-point instruction in the program code with a call to a type thunking stub. The thunk function bitcasts the operands into integers and jumps to the corresponding runtime library function. At execution time, these integer-only runtime functions compute all floating-point operations.

Interface. For basic functionality, the standard interface uses a strict subset of the SoftFloat [24] library. SoftFloat features a set of functions which take floats and doubles bitcast to unsigned integers and return bitcast results. All other soft floating-point libraries require small custom SoftFloat interface adapters. This standardization simplifies instruction rewriting with a single target interface.

LLVM instructions are rewritten as function calls to their SoftFloat counterparts. Table 1 lists the functions which replace LLVM instructions for symbolic interpretation. The instructions encode arithmetic, comparisons, and rounding which are handled by the soft floating-point functions.

Floating-point operation handlers are stored as interchangeable libraries for the interpreter. Depending on the emulation code, each library is compiled from C to LLVM bitcode (a binary representation of an LLVM assembly listing).

A bitcode library is native to the KLEE LLVM machine but can not support hand-coded assembly which is found in some soft floating-point implementations.

Software Encodings. Floating-point unpacking policies are either selective or unselective. SoftFloat selectively masks components out as needed to local variables. Both `bsdhppa` and `bsdppc`, on the other hand, completely unpack floating-point values into a data structure before every operation and repack the result into the IEEE-754 format. Each representation is acceptable depending on the circumstance. SoftFloat has a single compilation unit; the representation likely benefits from interprocedural analysis. BSD has multiple compilation units; mask calculations like SoftFloat's would repeat at function boundaries.

Operating System Handlers. The runtime uses off-the-shelf operating system code ported from one Linux and two BSD floating-point emulators. The soft floating-point implementations in operating systems are often quite good and at least heavily tested. This is because on several machine architectures (e.g., x86, ARM, MIPS), an operating system may be expected to emulate a hardware floating-point unit through software. Correctness and reproducibility demand that the software emulation closely matches hardware, hence operating systems should have accurate soft floating-point implementations.

In many cases, soft floating-point code can be compiled into LLVM bitcode for the runtime bitcode library. The floating-point emulation layer is intended to run on the target operating system architecture and is usually written in C. After adjusting a few header files, the code can be compiled independently into a bitcode library. Handlers accelerated with assembly code (e.g., x86 Linux, ARM BSD), however, must compile to native machine code.

An operating system's native emulation mechanism traps and emulates missing floating-point instructions. Whenever a user program issues a floating-point instruction, control is trapped and vectored to the operating system. The trapped instruction is decoded into a soft floating-point computation; the computation uses only integer instructions and stores the result to the machine state. Finally, control returns to the next program instruction. Since the symbolic executor rewrites floating-point instructions instead of trapping, the executor bypasses the decoding logic and directly calls the floating-point operations.

Library-specific SoftFloat glue code translates internal floating-point calls to the standardized SoftFloat interface. The internal functions never trap or decode instructions, so those stages are ignored and inaccessible. Porting requires relatively few lines of code; glue code for `linmips`, `bsdhppa`, and `bsdppc` is between 50 and 150 lines of C source.

4 Operation Integrity

Once a floating-point library is in place, it is possible to test the library. Test cases are gathered by symbolically executing a binary program with a given soft floating-point library for each floating-point operation. Cross-checking intermediate path data with the JIT over the host's hardware floating-point unit detects interpreter and soft floating-point inconsistencies. Pooling tests by operation across all libraries addresses the problem of single library underspecification; the JIT, with soft floating-point libraries and without, is checked against hardware execution of the binary on all test cases. Every library disagrees with hardware; some patterns emerge as common pitfalls. Finally, significant library coverage confirms the thoroughness of testing.

When comparing soft floating-point libraries with hardware, it is useful to distinguish between *consistency* and *verification*. If tests derived from a library $\mathcal{L}$ all match hardware, then $\mathcal{L}$ is *consistent* with hardware under a symbolic executor; for every library path there is a test case which matches hardware. When $\mathcal{L}$ is consistent on an operation $\circ$ it is $\circ$-*consistent*. Verification against hardware is stronger than consistency: all possible inputs for $\mathcal{L}$ must match hardware. Consistency without verification arises when an underspecified program misses edge cases which describe inconsistent test cases. Consequentially, tests from a library $\mathcal{L}^*$ may induce a mismatch on a consistent but underspecified library $\mathcal{L}$.

The soft floating-point code is tested in two phases. In the first phase, operation programs are symbolically executed to produce test cases for each library. To determine consistency, the symbolic executor is *cross-checked* with the LLVM JIT's machine code through a log replay mechanism that compares intermediate concrete values. If the emulation is wrong, symbolic interpretation diverges from the JIT values, potentially causing false positives and negatives with respect to native execution. In the second phase, to handle underspecification, operations are *cross-tested* on a test case pool and checked against an Intel Core2 processor. Further, the pool tests the JIT, which compiles floating-point LLVM instructions to machine code, and raises several translation errors.

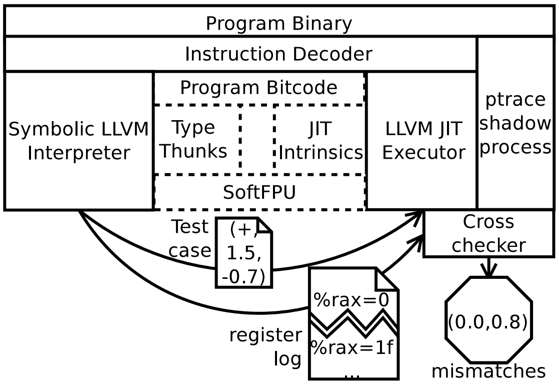


Fig. 2. Generating tests and cross-checking soft floating-point for a binary program

Figure 2 illustrates the process of cross-checking floating-point symbolic interpretation with a JIT and native hardware. To generate test cases, an x86-64 program binary is symbolically executed with KLEE and a soft floating-point bitcode library. The symbolic executor emits a register log file which cross-checks on replay with the LLVM JIT engine using native floating-point instructions (Interpreter $\times$ JIT). Test case values are inputs for hardware cross-checking (JIT $\times$ Hardware); the soft floating-point JIT is cross-checked, basic code block by basic block, against the registers from native execution of the binary program.

4.1 Gathering Test Cases

Each floating-point operation is modeled through a distinct test program binary. Every operation program is compiled from a small C file into an x86-64 Linux program that reads its IEEE-754 format operands from the standard input. Binary programs, instead of C source or LLVM bitcode, model operations to avoid compilation artifacts [36] which may interfere with native hardware evaluation of the program. Likewise, the operation programs test the machine code interface of the symbolic executor with compiled C, so some functionality is masked or inaccessible. This masking is particularly noticeable for the remainder (%) program which must use the math library `fmod` function because the C language only defines integer modulus operations.

Symbolic execution of an operation program reads operands from a *symbolic* standard input stream. When the symbolic executor completes a path, it creates a test case by selecting a feasible input bit-string which satisfies the path constraints imposed by the soft floating-point runtime library. Feeding the bit-string into the operand program reproduces the floating-point library path.

Table 2 lists the number of test cases produced by the symbolic executor for each floating-point library. Test programs are split by precision: 32-bit single-precision (f32) and 64-bit double-precision (f64). The number of test cases for a completed operation is a strict upper bound on the number of states that will fork on a floating-point instruction in a symbolically executed program. Implementation differences lead to unique test case counts for each operation across libraries; some libraries fork more (`bsdppc`) than others (`softfloat`).

A complete symbolic execution of an operation test program exhausts all paths. Every feasible path becomes a test input (e.g., pairs of floating-point values for binary operations) which satisfies the path constraints. Path exhaustion can be costly, however, so each program runs for a maximum time of one hour with a two minute solver timeout in order to enforce a reasonable limit on total execution time; most operations ran to completion. Relatively complicated operations, such as division, timed out across all libraries. The symbolic executor is sensitive to the branch organization of the library code so some simpler operations (e.g., + for `bsdppc` and `linmips`) time out from excessive state forking.

Table 2. Test cases from symbolic execution. Operations with † and ‡ timed out exploring paths and solving queries respectively.

Op.	**bsdhppa**		**bsdppc**		**linmips**		**softfloat**		**softgun**	
	f32	f64	f32	f64	f32	f64	f32	f64	f32	f64
$+$	122	1017	$1670^{\dagger}$	$2016^{\dagger}$	3741	$6523^{\dagger}$	99	99	458	868
-	122	1017	$1615^{\dagger}$	$1717^{\dagger}$	3738	$6480^{\dagger}$	99	99	458	868
$*$	$3700^{\ddagger}$	$520^{\dagger\ddagger}$	$368^{\dagger\ddagger}$	$349^{\dagger\ddagger}$	$1945^{\ddagger}$	$2235^{\dagger\ddagger}$	51	51	$388^{\ddagger}$	$582^{\dagger\ddagger}$
$/$	$6109^{\dagger}$	$6268^{\dagger}$	$2932^{\dagger}$	$3520^{\dagger}$	$4694^{\dagger}$	$5268^{\dagger}$	$132^{\ddagger}$	$81^{\dagger\ddagger}$	$6373^{\dagger}$	$6639^{\dagger}$
%	$3247^{\dagger}$	$3359^{\dagger}$	2	$3680^{\dagger}$	$3156^{\dagger}$	$3397^{\dagger}$	$2900^{\dagger}$	$3010^{\dagger}$	$3057^{\dagger}$	$3394^{\dagger}$
$<$	28	32	$1341^{\dagger}$	$1661^{\dagger}$	91	91	7	7	11	13
$\leq$	34	38	$2034^{\dagger}$	$2353^{\dagger}$	91	91	7	7	11	13
$=$	2890	42	$1359^{\dagger}$	$1140^{\dagger}$	2905	91	125	14	1061	18
$\neq$	2890	42	$1402^{\dagger}$	$1413^{\dagger}$	2905	91	125	14	1061	18
$\leftrightarrow$	29	31	29	91	29	22	8	14	28	27
$\rightarrow$i32	81	67	68	69	65	20	17	12	64	15
$\rightarrow$i64	142	142	$171^{\dagger\ddagger}$	$145^{\dagger\ddagger}$	97	82	39	34	201	211
$\leftarrow$i32	14	6	75	34	51	35	5	4	72	34
$\leftarrow$i64	118	46	$1466^{\dagger}$	491	191	89	14	8	477	119
$\sqrt{x}$	$3451^{\dagger}$	$2979^{\dagger}$	$2583^{\dagger}$	$4500^{\dagger}$	$4856^{\dagger}$	$3819^{\dagger}$	$32^{\ddagger}$	$25^{\ddagger}$	$92^{\ddagger}$	$1355^{\dagger\ddagger}$

4.2 Cross-Checking for Consistency: Interpreter $\times$ JIT

Symbolically executing an operation $\circ$ produces a set of test inputs which exercise distinct paths through the floating-point emulation library. Ideally, the symbolic interpreter exhausts all paths on $\circ$, leading to a test case for every possible path on $\circ$. The test cases are replayed concretely on the LLVM JIT and cross-checked with the symbolic interpreter's concrete values to find bugs in the symbolic interpeter. Testing with cross-checking determines consistency; when all of $\circ$'s test cases for a library cross-check as matching, the library is $\circ$-consistent.

The symbolic interpreter is a custom LLVM interpreter which may diverge from bitcode semantics. To find divergences, the interpreter's soft floating-point library results (in emulated hardware registers) are checked against natively executed LLVM JIT machine code for bit-equivalence at every dispatched VEX-decoded instruction block. These checks replay a test case on the JIT and cross-check with the interpreter's concrete register log.

The number of failures for symbolically generated test cases is given in Table 3. There are three reasons for failure to cross-check: 1) the path terminated early (log runs out), 2) the floating-point library is wrong, or 3) the JIT engine is wrong. Failure to complete paths is demonstrated by `softfloat` division and `linmips` multiplication. All libraries fail cross-checking as highlighted in Table 4. It is likely the libraries have never been systematically cross-checked, so inconsistency is expected. Furthermore, JIT engine errors arise in $\sqrt{x}$ and % for `bsdppc` and `softfloat`, but are discussed in the next section because they require systematic hardware cross-checking for confirmation.

Table 3. Failures on JIT register log cross-checking from library consistency tests

Op.	**bsdhppa**		**bsdppc**		**linmips**		**softfloat**		**softgun**	
	f32	f64	f32	f64	f32	f64	f32	f64	f32	f64
$+$	1	1	0	76	0	0	0	0	0	0
-	1	1	0	5	0	0	0	0	0	0
$*$	2	30	29	28	0	8	0	0	0	18
$/$	2	1	2	0	0	0	0	29	1	0
%	7	3	0	828	30	2	13	3	67	207
$<$	0	0	0	148	0	0	0	0	1	2
$\leq$	0	0	0	368	0	0	0	0	2	1
$=$	0	0	174	161	0	0	0	0	718	12
$\neq$	0	0	233	213	0	0	0	0	718	12
$\leftrightarrow$	0	0	6	0	1	2	0	0	1	0
$\rightarrow$i32	2	2	12	0	0	0	2	0	0	0
$\rightarrow$i64	2	2	86	5	0	0	2	0	169	152
$\leftarrow$i32	0	0	30	19	0	0	0	0	0	0
$\leftarrow$i64	0	1	312	76	0	0	0	0	0	0
$\sqrt{x}$	9	92	8	4	5	1	8	10	84	18

4.3 Cross-Testing for Underspecification Bugs

A consistent floating-point library may appear correct with path-exhaustive testing but it is still unverified. Consistency merely demonstrates interpreter and JIT equivalence; it is disconnected from hardware ground-truth. Some components could be underspecified, leading to a false confidence. Underspecification may stem from the library (and therefore its tests) partially describing IEEE-754 semantics or the decoder mistranslating machine instructions. Pooling all tests for all discovered library paths and testing concrete JIT replay against native processes, on the other hand, finds bugs from underspecification and mistranslation.

Pooled Tests. Cross-checking is limited to detecting mismatches on known inputs; the symbolic executor must provide the input test cases. If a floating-point library is *underspecified*, then symbolic execution may not generate a test case for a hardware mismatch. An underspecified library is incorrect despite being consistent. In fact, many operations cross-check as consistent; `softfloat` and `linmips` seem to perfectly convert floating-point to and from integers because the libraries are {$\rightarrow$i32,$\rightarrow$i64, $\leftarrow$i32, $\leftarrow$i64}-consistent. However, these consistent operations are not correct but underspecified.

The underspecification problem can be partially mitigated by cross-testing across libraries. All tests are pooled and hardware cross-checked on all floating-point libraries. The test pool is applied to each library to cover values or conditions missed by underspecification.

Figure 3 illustrates the distinct value coverage for each library. There are two ways to account for the uneven test case distribution among the libraries. One,

Table 4. Selected mismatches from soft floating-point library consistency tests

Library	Operation	Soft FP	Hardware FP
`bsdhppa`	$\infty * 0.0$	`NaN`	`-NaN`
`bsdppc`	1.1125...6e-308 + 1.1125...7e-308	5e-324	2.2250...1e-308
`linmips`	`0x7ff0000001000000 (NaN)` $\rightarrow$f32	`0x7fbfffff (NaN)`	`0x7fc00000 (NaN)`
`softfloat`	`NaN` $\rightarrow$i32	`0x7fffffff`	`0x80000000`
`softgun`	0.0 / 0.0	0.0	`-NaN`

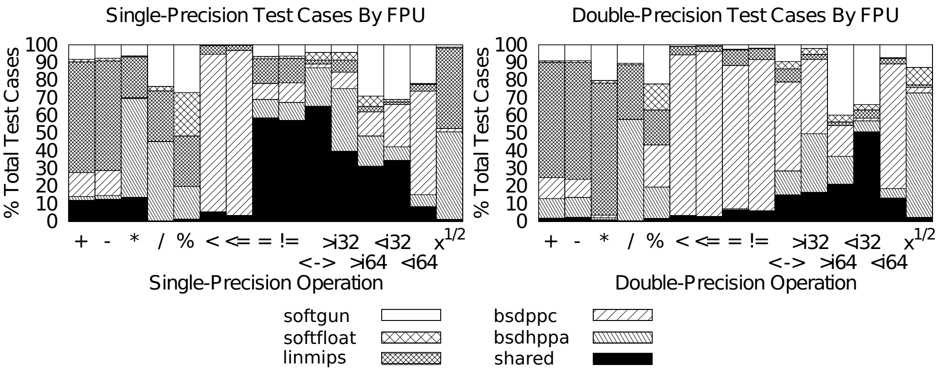


Fig. 3. Floating-point test case distribution

interpretations could follow the same model, but the states are sliced differently. Two, given distinct models, the extra test cases may cover a divergence between library and hardware floating-point algorithms. Ideally, the test cases would uncover no divergences because no library would be underspecified.

In total, there were 103624 distinct test cases. The test count is 34 orders of magnitude better than brute force testing ($(6(2^{32} + 2^{64}) + 9(2^{64} + 2^{128}))$ tests). However, so many tests may be relatively inefficient; one hand-designed suite [30] for $\exp(x)$ uses 2554 tests. When test count is a concern (e.g., tests are expensive), non-exhaustive execution can give useful results. We found randomly dropping forked states on hot branches still covered incorrect values for all libraries with 19560 distinct tests.

Checking JIT $\times$ Hardware. Table 5 shows cross-checking errors found from cross-testing with a pool of foreign test cases. The JIT is cross-tested with a `ptrace`d native shadow process; for every dispatched decoded instruction block, `ptrace` hardware registers are checked against the JIT's emulated registers. A single bit difference is an error for a test case. Consistent operations, such as `softfloat`'s $-$, are shown to be incorrect and underspecified by the test pool.

Applying the tests to the JIT engine, which is never symbolically executed, and comparing the registers with hardware execution revealed bugs on interesting edge cases. For instance, $\sqrt{x}$ on a negative single-precision value returns `NaN` in the JIT, but `-NaN` for hardware and `softfloat`. Errors in the JIT from translating machine

Table 5. Hardware cross-check errors from all distinct tests

Op.	**bsdhppa**		**bsdppc**		**linmips**		**softfloat**		**softgun**		**JIT**	
	f32	f64	f32	f64	f32	f64	f32	f64	f32	f64	f32	f64
+	603	135	7	8034	45	90	6	12	80	92	0	0
-	623	109	47	5354	45	63	6	9	62	81	0	0
*	50	53	8	1295	21	23	8	7	23	28	0	0
/	56	52	2	831	32	28	2	4	37	41	0	0
%	176	123	134	13	176	58	9	7	4263	4638	35	3
$<$	0	0	0	270	0	0	0	0	52	402	0	0
$\leq$	0	0	0	405	0	0	0	0	72	609	0	0
$=$	0	0	650	7	0	0	0	0	4665	125	0	0
$\neq$	0	0	669	6	0	0	0	0	4736	204	0	0
$\leftrightarrow$	5	84	49	19	4	7	0	0	2	4	0	0
$\rightarrow$i32	40	32	86	38	25	26	40	31	25	26	0	0
$\rightarrow$i64	153	76	304	121	24	27	47	36	257	264	0	0
$\leftarrow$i32	147	75	147	75	0	0	0	0	0	0	0	0
$\leftarrow$i64	1668	650	1671	651	55	0	55	0	55	0	55	0
$\sqrt{x}$	36	6	39	18	36	0	0	0	36	6	36	6

x86-64 Assembly

```
cvtsi2ss   %rbx, %xmm2    # i64 -> f32
movd       %xmm2, %rcx    # get f32
```

Vex IR

```
t1 = 64to32(And64(GET:I64(216), 0x3:I64)) # get rounding mode
t2 = GET:I64(40) # get rbx
PUT(288) = F64toF32(t1, I64StoF64(t1, t2)) # i64 -> f32
PUT(24) = GET:I64(288)  # f32 -> rcx
```

Fig. 4. Mistranslation of the `cvtsi2ss` instruction in the VexIR

code (a known problem for machine code interpreters on integer workloads [19,33]) to LLVM bitcode appear across all libraries (e.g., f32←i64). Mismatches which appear solely for the JIT could be due to instruction selection.

For an in-depth example, Figure 4 reproduces the code involved for a f32←i64 conversion error. The operation program begins as x86-64 machine code which is translated by the VEX decoder into VEX IR, then from VEX IR into LLVM bitcode; if VEX mistranslates the machine code, the symbolic interpretation will be wrong. In the case of f32←i64, the x86-64 instruction `cvtsi2ss` converts a 64-bit signed integer to a single precision floating-point number. The corresponding VEX IR converts the signed integer into a double precision number, then to single precision. This induces rounding errors (confirmed by the VEX maintainer) that cause certain inputs (e.g., `rbx` = 72057598332895233) to evaluate one way natively (7.20576e+16) and another way through VEX (7.2057594e+16).

4.4 Common Pitfalls

Some effort was put into improving the library code's cross-checking results before finalizing the data. The `softfloat` and `linmips` libraries were intended to be consistent whereas the consistency of `bsdhppa` and `bsdppc` was a lower priority. Most improvements concentrated on a few frequent problems.

Endianness. Architecture byte-order can conflict with the hardware byte-order. The PA-RISC (`bsdhppa`), PowerPC (`bsdppc`), and MIPS (`linmips`) architectures are big-endian but the host x86-64 machine is little-endian. MIPS is dual-endian so `linmips` has a `#define` to enable little-endian mode. For `bsdhppa`, the glue code must swap double-precision operands and results. `bsdppc`, evolved from SPARC and m68k code bases, is staunchly big-endian. For instance, a 32-bit function for converting double precision values expects the most significant half of a 64-bit value as its 32-bit return result, a big-endian convention.

Default `NaN`. Certain operations, such as division by zero, produce a default "quiet" `NaN`. Unfortunately, hardware is free to deterministically choose a `QNaN` from 2^{23} bit-patterns. Both single and double precision `QNaN`s differed from the host machine for every library. The single-precision `QNaN` was `0x7fffffff`, as opposed to x86-64 hardware which uses `0xffc00000`. Manual inspection of the Linux x87 emulator confirmed the values on x86-64 matched the expected `QNaN`.

`NaN` operands. The x86-64 floating-point unit encodes extra diagnostic information into the mantissa of its `NaN`s which disagrees with emulation. There are numerous ways to mishandle a `NaN` operation so the bits do not match. For operations between `NaN`s, a signaling `NaN` would sometimes be converted into the wrong `QNaN`. Arithmetic between a `NaN` and number would use the default `NaN`, missing the mantissa bits. Likewise, operands returning the left-hand `NaN` in hardware instead used the library default `NaN`. Although this extra information is optional, it is still desirable to stay bit-identical to the host hardware.

4.5 Coverage

Operation test cases raise plenty of mismatches but the depth of testing remains unclear. Code coverage for each library is a simple metric for overall testing quality; high coverage implies thorough testing. Figure 5 shows the coverage of each floating-point implementation from symbolic execution. The instruction coverage percentages are calculated from visited functions. The set of instructions is limited to visited functions because soft floating-point libraries often have additional features which are inaccessible through the interpreter (e.g., trapped instruction decoding). Total covered instructions gauges the complexity, although not necessarily the correctness, of the library implementation.

Between 79%–95% of instructions were covered by test cases for each library, which leaves 5%–21% of instructions uncovered. There are several justifiable reasons for missing instructions. All libraries support all rounding modes but only round-nearest is tested because it is the default mode. Compiler optimizations mask paths; when converting x to floating-point, one optimization tests if x is 0 to avoid the floating-point instruction, leaving the 0 library path unexplored.

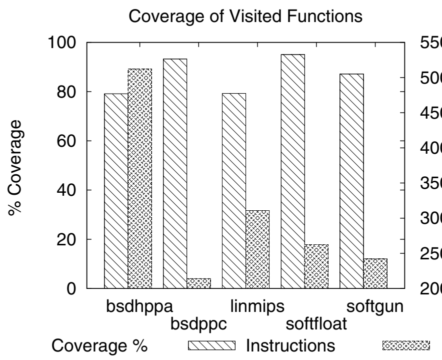


Fig. 5. Soft floating-point library code coverage and total covered instructions

Finally, compiled-in assertions, such as checking for bad type tags, often remain uncovered because the code never contradicts the assertion predicate.

5 Floating-Point SMT Solvers

This section evaluates the accuracy of floating-point solvers with respect to floating-point hardware. A floating-point solver decides the satisfiability of formulas over a theory of floating-point and therefore must at least encode floating-point semantics like those found in soft floating-point libraries. Prior work [41] suggests testing floating-point solvers with randomly generated conformance formulas. Unlike randomized conformance queries, test cases derived from soft floating-point target interesting edge cases defined by the emulation code. Despite the importance of accuracy, testing a selection of solvers reveals divergent results in every solver. Furthermore, each floating-point solver has implementation quirks which impede testing with all operations and values.

Several freely available contemporary floating-point solvers support a theory of IEEE-754 floating-point. These solvers include mathsat5-2.8 [22], sonolar-2013-05-15 [39], and Z3 [15] (current stable and unstable FPA versions from the git repository). Such specialized floating-point solvers back the only complete symbolic execution alternative to soft floating-point. However, these solvers only occasionally conform to a standard interface, have complicated internals and, despite formal proofs of correctness, are clearly wrong in many cases.

SMTLIB2-FPA [42] (a proposed standard) conformance tests [41] provide a useful baseline test suite for SMTLIB2-FPA compliant solvers (mathsat and Z3). The conformance tests cover a range of features in SMTLIB2-FPA. These tests exercise the front-end and floating-point theory for a floating-point solver. A solver implements SMTLIB2-FPA in its front-end by translating floating-point arithmetic (FPA) theory primitives into an internal floating-point representation. The tests were generated from the reference implementation with random floating point numbers for a total of 20320 SMTLIB2-FPA queries.

```
(set-logic QF_FPA)
(set-info :status sat)
(assert (= (/ roundNearestTiesToEven
         ; a = 4.96875
         ((_ asFloat 8 24) (_ bv0 1) (_ bv2031616 23) (_ bv129 8))
         ; b = 1.9469125e-38
         ((_ asFloat 8 24) (_ bv0 1) (_ bv5505024 23) (_ bv1 8)))
     ; r = 2.5521178e+38
     ((_ asFloat 8 24) (_ bv0 1) (_ bv4194304 23) (_ bv254 8))))
(check-sat)
; a / b = 2.5521178e+38 is sat, but model claims a/b=plusInfinity
```

Fig. 6. A Z3 test case query for checking a single-precision division result

Table 6. Conformance and library test cases applied to several FPA solvers

Solver	Conformance		Library Tests		
	Pass	**Fail**	**Pass**	**Fail**	**Unknown**
`mathsat`	14487	5833	10096	0	2455
`sonolar`			10699	482	0
`z3-stable`	20023	297	7081	2931	1076
`z3-fpa`	20090	230	9996	16	1076

The automatically generated test cases from soft floating-point libraries are novel in that they work as semantically derived tests for third-party floating-point solvers. Each test includes an operation and operands (e.g., $(+\ a\ b)$) and a scalar result r computed through hardware. Each solver is tested against the hardware result by checking that the operation feasibly evaluates to r with a bit-vector equality query as in Figure 6. The library tests are based on the smaller fork-inhibited data set from Section 4.3 to reduce testing overhead. These tests are simple satisfiability tests on concrete expressions; they neither impose symbolic constraints on operands nor examine counter-examples.

Table 6 lists the test results for the floating-point solvers. Even though the tests are shallow, every solver fails some test (fail) or gives no answer after five minutes or refusing certain inputs (unknown). Each row only has a subset of the tests because the front-end and library interfaces lack particular operations (e.g., $\neq$). Overall, the rate of failure indicates these solvers are currently more appropriate for domain-specific applications than general program testing.

Each solver has its own quirks. mathsat's front-end accepts the most recent SMTLIB2-FPA proposal but misses a rounding mode in the conformance tests. For concrete tests mathsat is consistent but rejects `NaN` inputs and often times out on division operations. Sonolar only supports floating-point with library bindings. The Z3 solver accepts obsolete SMTLIB2-FPA and lacks some type conversions. Furthermore, the stable branch of Z3 is nearly a year old; the current unstable Z3 FPA branch is an improvement but still diverges from hardware.

Table 7. Bugs found in Linux programs following floating-point computation

Bug Type	**Programs**
Divide By Zero	26
Bad Write	57
Bad Read	259
Total Programs	314

6 Bugs in Linux Programs

The previous sections focus on custom testing with soft floating-point; here, the symbolic executor is applied to a large general program set. Test cases are collected by symbolically executing program binaries belonging to two Linux distributions. Floating-point arithmetic appears in hundreds of program faults from these test cases. However, the influence of floating-point operations can be subtle and independent of the crash. Additionally, few paths use the symbolic floating-point capabilities but the overhead from concrete evaluation with soft floating-point is often negligible compared to built-in concrete evaluation.

Test cases were collected from five minutes of symbolic execution on program binaries from Ubuntu 13.10 (x86-64) and Fedora 19 (x86) using the SoftFloat library. The symbolic executor analyzed 27795 binaries total with 4979 binaries raising some kind of error. To isolate floating-point tests, only paths covering floating-point instructions on replay (837 test cases) are considered.

Table 7 presents a summary of flagged floating-point programs. Test cases are classified as errors by the type of program fault they cause. The errors are validated with a replay mechanism based on the JIT (568 test cases); fault-free test cases in the JIT replay are ignored. The largest class, bad memory reads, frequently accessed lower addresses which presumably stem from `NULL` pointers.

Floating-point is often independent of the actual bug; programs such as `gifrsize`, `scs2ps`, and `pngcrush` all access `NULL` pointers solely through integer constraints. Regardless, floating-point numbers may subtlety influence symbolic integer values. The `unicoverage` program (crashes on a buffer overflow) lends an example expression: `100.0*nglyphs/(1+cend-cstart)`. Terms `cend` and `cstart` are symbolic integers (read from `scanf`) and `nglyphs` is a concrete integer. The floating-point 100.0 term coerces the symbolic integer expression into a double precision floating-point value. The floating-point multiplication therefore imposes floating-point constraints (from the i32→f64 operation) on integer-only terms. Curiously, manual inspection of many reports yielded no direct floating-point crashes (e.g., a dereference with a floating-point index) but this may be a symptom of the brief symbolic execution time per program.

The majority of floating-point test cases rely solely on concrete floating-point data. Only 94 programs (18%) forked on soft floating-point library code and hence processed any symbolic floating-point values at all. Programs which process only concrete floating-point data incur overhead from dispatching extra instructions for floating-point emulation. The instruction overhead from emulating concrete floating-point with integer code, compared to the default KLEE concrete floating-point dispatch, is negligible. Soft floating-point tests incur 242058 extra instructions on average (1.04× overhead) with a 135474 instruction standard deviation (1.61× overhead) and 613 instruction median (1.0007× overhead). Floating-point heavy programs skew the average: a program for processing triangulated meshes, `admesh`, suffered the maximum instruction overhead of 6.98×, followed by `bristol`, an audio synthesizer emulator, with 2.68× overhead.

7 Conclusion

The best software analysis tools must soundly model floating-point. Floating-point as a runtime library is perhaps the simplest worthwhile way to model high-quality floating-point semantics in a symbolic binary executor. This quality comes from soft floating-point being testable within reason through a combination of symbolic execution and cross-checking against hardware. Integer-only symbolic execution produces concrete test files with floating-point information which can be directly confirmed by hardware. If important tests are missed from underspecification, they may be found by testing with multiple floating-point libraries. Finally, although the library testing is incomplete in some cases, the results have demonstrated that a symbolic soft floating-point unit is sufficient for finding many verifiable test cases for bugs in commodity binary programs.

Acknowledgements. Dawson Engler, anonymous reviewers, and shepherding from Sriram Sankaranarayanan provided valuable feedback which improved the quality of this paper. This work was supported in part by the US Air Force through contract AFRL-FA8650-10-C-7024 and by DARPA award HR0011-12-2-009. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the Air Force.

References

1. Alglave, J., Donaldson, A.F., Kroening, D., Tautschnig, M.: Making software verification tools really work. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011)
2. Bagnara, R., Carlier, M., Gori, R., Gotlieb, A.: Symbolic path-oriented test data generation for floating-point programs. In: Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation, p. 10. IEEE Press, Luxembourg City (2013)
3. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, pp. 549–560. ACM, New York (2013)
4. 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: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLD 2003), June 7-14, pp. 196–207. ACM Press, San Diego (2003)

5. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16(2), 97–121 (2006)
6. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD, pp. 69–76. IEEE (2009)
7. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 463–469. Springer, Heidelberg (2011)
8. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008, pp. 209–224 (2008)
9. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: ASPLOS 2011, pp. 265–278 (2011)
10. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
11. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic crosschecking of floating-point and SIMD code. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys 2011, pp. 315–328. ACM, New York (2011)
12. Conchon, S., Melquiond, G., Roux, C., Iguernelala, M.: Built-in treatment of an axiomatic floating-point theory for SMT solvers. In: Fontaine, P., Goel, A. (eds.) SMT 2012. EPiC Series, vol. 20, pp. 12–21. Easy Chair (2013)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, pp. 238–252. ACM, New York (1977)
14. Davis, E.: Constraint propagation with interval labels. Artificial Intelligence 32(3), 281–331 (1987)
15. 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)
16. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
17. Godefroid, P., Kinder, J.: Proving memory safety of floating-point computations by combining static and dynamic program analysis. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 1–12. ACM, New York (2010)
18. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network Distributed Security Symposium (2008)
19. Godefroid, P., Taly, A.: Automated synthesis of symbolic instruction encodings from I/O samples. In: PLDI, pp. 441–452 (2012)
20. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys 23, 5–48 (1991)
21. Goubault, É., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 18–34. Springer, Heidelberg (2006)
22. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding floating-point logic with systematic abstraction. In: Cabodi, G., Singh, S. (eds.) FMCAD, pp. 131–140. IEEE (2012)
23. Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 76–92. Springer, Heidelberg (2009)
24. Hauser, J.: SoftFloat-2b (2002), `http://www.jhauser.us/arithmetic/SoftFloat.html`

25. IEEE Task P754: ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic (August 1985)
26. Ivancic̀, F., Ganai, M.K., Sankaranarayanan, S., Gupta, A.: Software model checking the precision of floating-point programs. In: Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), pp. 49–58. IEEE (2010)
27. Kahan, W.: Implementation of algorithms (lecture notes by W. S. Haugeland and D. Hough). Technical Report 20 (1973)
28. Karrer, J.: Softgun – the embedded system simulator (2013), `http://softgun.sourceforge.net`
29. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19, 385–394 (1976)
30. Kuliamin, V.V.: Standardization and testing of implementations of mathematical functions in floating point numbers. Programming and Computer Software 33(3), 154–173 (2007)
31. Lakhotia, K., Tillmann, N., Harman, M., de Halleux, J.: FloPSy - search-based floating point constraint solving for symbolic execution. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 142–157. Springer, Heidelberg (2010)
32. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 215–224. ACM, New York (2012)
33. Martignoni, L., McCamant, S., Poosankam, P., Song, D., Maniatis, P.: Path-exploration lifting: hi-fi tests for lo-fi emulators. In: ASPLOS 2012, pp. 337–348. ACM, New York (2012)
34. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D.A. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
35. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009, pp. 67–82. USENIX Association (2009)
36. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Trans. Program. Lang. Syst. 30(3), 12:1–12:41 (2008)
37. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI 2007, pp. 89–100 (2007)
38. O'Leary, J., Zhao, X., Gerth, R., Seger, C.J.H.: Formally verifying IEEE compliance of floating-point hardware. Tech. rep., Intel Technical Journal (First quarter 1999)
39. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)
40. Păsăreanu, C., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. International Journal on Software Tools for Technology Transfer (STTT) 11, 339–353 (2009)
41. Rümmer, P.: Preliminary SMT-FPA conformance tests (2010), `http://www.cprover.org/SMT-LIB-Float/`
42. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: Informal Proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland (2010)

Monitoring Parametric Temporal Logic

Peter Faymonville[1], Bernd Finkbeiner[1], and Doron Peled[2]

[1] Fachrichtung Informatik
Universität des Saarlandes, Germany
[2] Department of Computer Science
Bar Ilan University, Israel

Abstract. Runtime verification techniques allow us to monitor an execution and check whether it satisfies some given property. Efficiency in runtime verification is of critical importance, because the evaluation is performed while new events are monitored. We apply runtime verification to obtain *quantitative* information about the execution, based on linear-time temporal properties: the temporal specification is extended to include parameters that are instantiated according to a measure obtained at runtime. The measure is updated in order to maintain the best values of parameters, according to their either maximizing or minimizing behavior, and priority. We provide measuring algorithms for linear-time temporal logic with parameters (PLTL). Our key result is that achieving efficient runtime verification is dependent on the determinization of the measuring semantics of PLTL. For *deterministic PLTL*, where all disjunctions are guarded by atomic propositions, online measuring requires only linear space in the size of the specification and logarithmic space in the length of the trace. For *unambiguous PLTL*, where general disjunctions are allowed, but the measuring is deterministic in the truth values of the non-parametric subformulas, the required space is exponential in the size of the specification, but still logarithmic in the length of the trace. For full PLTL, we show that online measuring is inherently hard and instead provide an efficient offline algorithm.

1 Introduction

While verifying the complete behavior of a system (e.g., using model checking) is certainly desirable, it is not always possible, as its internal structure is not always given, or its state space is prohibitively large. Runtime verification analyzes the ongoing execution of a system against a given specification, written for example in Linear Temporal Logic (LTL), based on monitoring its externally measurable events. The challenge in runtime verification is to provide an efficient algorithm that can perform its required *updates* between any two successive monitored events.

We study here the runtime monitoring of a system with respect to temporal properties, expressed using LTL, where (discrete time) duration counters are added to the subformulas. The runtime verification reports not only about the conformance between the currently monitored sequence and the specification, but also provides numerical values that bound the duration of the scope of subformulas on the checked execution from either above or below. For example, the specification $\Box(r \rightarrow \Diamond_{\leq x} g)$ computes the maximal response time x between a request r and a response g. We call this approach for runtime verification "runtime measuring" (as "model measuring" [2] is related to "model checking"). While runtime verification can check and alarm against unwanted situations, runtime measuring collects statistics on the system behavior.

Our monitoring approach is *declarative* as opposed to *operational*. In an operational approach [7,8], the measures are collected by explicitly specifying the initialization and update of counters that calculate the reported values. In a declarative approach, we attach parameters to temporal formulas to specify the measure we are interested in, and the monitoring algorithm takes care of finding parameter valuations such that the formula is satisfied. Suppose, for example, that we do not want to measure all response times, but are interested only in the *last* request that was successfully answered. The formula $\square(\bigcirc\Diamond(r\wedge\Diamond g)\vee(r\to\Diamond_{\leq x}g))$ uses the disjunction to filter out all requests before the last successful request: in every step, if the left disjunct holds, then the response time is irrelevant, because the disjunction is true for any value of x; the right disjunct thus only becomes relevant when the left disjunct is false, i.e., when there is no future request that is successfully answered.

Research on the runtime verification of LTL [6,8,10] distinguishes *online* algorithms that need to keep only some bounded amount of information about the trace seen so far, from *offline* algorithms that store the entire sequence for later evaluation. For runtime measuring, we also look for an online algorithm with reasonable space consumption. Since measuring necessarily entails updating counters, an algorithm that requires logarithmic space in the length of the trace is acceptable, while an algorithm that requires linear space in the length of the trace, such as an offline algorithm, is in general not practical. We show that the complexity of runtime measuring depends on the disjunctive characterization of the LTL specification; that is, when the formula contains a disjunction, or some subformula that can be satisfied in different ways (e.g., $\varphi\,\mathcal{U}\,\psi$ may be satisfied when the first ψ holds, but also when φ continues to hold until some subsequent occurrences of ψ). Due to this disjunction characteristics, counting results are not uniquely defined. Consider a sequence of length n where a, b and c happen in each event. The formula $(a\,\mathcal{U}_{\leq x}\,(b\,\mathcal{U}_{\leq y}\,c))$ can obtain any natural number for the parameters x and y such that $x+y=n$. One way to obtain unique measures is to impose a priority order among the parameters and seek for the best (minimal) results according to the lexicographic ordering. We show that for this case, an online measuring algorithm with logarithmic memory in the length of the trace is impossible.

An alternative approach to obtain unique measures is to modify the logic. We present two variations of PLTL that not only provide unique measures, but also have efficient online measuring algorithms. *Deterministic PLTL* allows, like deterministic LTL [11], only guarded disjunctions of the form $((p\wedge\psi_1)\vee(\neg p\wedge\psi_2))$, where p is an atomic proposition. Runtime measuring of deterministic PLTL indeed requires only logarithmic memory. The drawback of deterministic PLTL is, however, its limited expressiveness: deterministic LTL can only express properties in the intersection of LTL and ACTL [11]. To eliminate this drawback, we introduce *unambiguous PLTL*, which maintains the full expressiveness of LTL. Instead of syntactically restricting the possible disjunctions, unambiguous PLTL only *disambiguates* the PLTL semantics with respect to the measuring. For example, in a disjunction $(\psi_1\vee\psi_2)$, we *only* measure ψ_2 if ψ_1 is is false under all possible parameter values.

Consider, for example, again the problem of measuring the maximal response time x between a request r and a response g. To express this measuring problem in deterministic PLTL, we use a disjunction guarded by r: $\Box(\neg r \vee (r \wedge \Diamond_{\leq x} g))$. This encoding relies on the fact that the condition that triggers the measuring, r, is an atomic proposition. If we modify the problem, as discussed earlier, to only measure the last successful request, then this is no longer possible, because the decision whether or not to measure the current request depends on the success of *future* requests. The modified measuring problem can thus no longer be expressed in deterministic PLTL. The PLTL specification $\Box(\bigcirc\Diamond(r \wedge \Diamond g) \vee (r \rightarrow \Diamond_{\leq x} g))$, discussed above, does, however, work for unambiguous PLTL. In unambiguous PLTL, the (unguarded) disjunction is allowed, and the right disjunct is evaluated whenever the left disjunct is false. Hence, the specification computes precisely the response time of the last successful request.

We obtain the following results: for deterministic PLTL, online measuring requires only logarithmic space in the length of the trace and linear space in the size of the specification. For unambiguous PLTL, the required space is exponential in the size of the specification, but still logarithmic in the length of the trace. For full PLTL, we provide an efficient offline algorithm, which can also be used as an online algorithm by keeping the trace seen so far in storage. This algorithm requires quasilinear space in the length of the trace. We also show that, in fact, no online measuring algorithm with logarithmic space in the length of the trace exists. Unambiguous PLTL thus appears to be the sweet spot in the trade-off between expressiveness and complexity.

Related Work. The synthesis of monitors for LTL is a well-studied problem, see [3,5,6,8,10,13]. The offline backwards runtime algorithm of Havelund and Rosu [6] calculates with each event the truth values of the subformulas according to subformula order. Thus, with each new monitored event, the calculation would be *linearly* related to both the length of the sequence so far and the size of the checked formula. The *testers* construction by Pnueli and Zaks [13] can be used to backwards assign values to variables representing subformulas in a compositional manner. In previous work of the second author together with Sankaranarayanan and Sipma [8,7], alternating automata are used to obtain efficient algorithms for runtime verification. The query language considered there extends LTL with functions that are executed along the trace in order to collect measures and more complicated statistics such as the average number of packet transmissions in a communication protocol. Unlike the declarative approach of this paper, the collection of measures and statistics is specified operationally, not in the form of parameters. A different type of parametric monitoring has been studied by Rosu and Chen [14]: They consider traces that contain events with parameter bindings. Such traces can be considered as several different traces merged together and the challenge for the monitoring algorithm lies in the efficient slicing of the trace.

2 Parametric Temporal Logic

Syntax. Parametric Temporal Logic (PLTL) [2] is an extension of linear-time temporal logic (LTL) [12] with parameterized operators, which measure the duration from the introduction of a temporal goal until it is satisfied. E.g., for a subformula of the form

$\varphi \,\mathcal{U}\, \psi$, we expect to measure the duration until ψ happens. We assign a parameter x together with a comparison operator to the subformula, e.g., $\varphi \,\mathcal{U}_{\leq x}\, \psi$, with the intended meaning that ψ should hold within *at most* x steps while φ holds. In contrast to LTL variants like Prompt-LTL [9], PLTL allows multiple parameters within a formula.

The *syntax* of PLTL is given, for a set of atomic propositions AP, with typical element p, and a set of parameter variables V as follows:

$$\psi ::= \mathit{true} \mid p \mid \neg\psi \mid (\psi \wedge \psi) \mid (\psi \vee \psi) \mid \bigcirc\psi \mid \Diamond\psi \mid \Box\psi \mid (\psi \,\mathcal{U} \psi) \mid (\psi \mathcal{R} \psi) \mid \Diamond_{\leq x}\psi \mid \Box_{\leq x}\psi$$

The operators $\mathcal{U}$-*until*, $\Diamond$-*eventually*, $\Box$-*always*, $\mathcal{R}$-release, $\bigcirc$-*next* are the usual temporal operators from LTL. In the two new parametric operators $\Diamond_{\leq x}\psi$ and $\Box_{\leq x}\psi$, x may be either a constant natural number or a parameter variable.

In addition to disjunction, conjunction, and negation, we also allow the usual derived Boolean connectives such as implication $\rightarrow$. In addition to $\Diamond_{\leq x}\psi$ and $\Box_{\leq x}\psi$, we also use the following derived parametric operators

$$\Diamond_{>x}, \Box_{>x}, \mathcal{U}_{\leq x}, \mathcal{U}_{>x}, \mathcal{R}_{\leq x}, \text{ and } \mathcal{R}_{>x},$$

where x is again a parameter or a natural number. We assume that each parameter variable occurs at most once. Let $\alpha : X \mapsto \mathbb{N} \cup \{\infty\}$ denote a value assignment for the parameters. Then $\alpha(x)$ is the integer value assigned to x by α. For simplicity, we set $\alpha(k) = k$ for $k \in \mathbb{N}$. We denote by $\alpha[k/x]$ the valuation that maps y to k if $y = x$ and to $\alpha(y)$ if $y \neq x$. Let $\alpha \setminus x$ be the valuation α, excluding the parameter x.

Semantics. In the following, we adapt the PLTL semantics to the finite traces observed during monitoring. We interpret a given PLTL formula ψ over a finite trace σ of events, numbered with nonnegative integers, where each event provides an interpretation to the Boolean propositions AP, i.e., $\sigma : \{0 \dots |\sigma| - 1\} \rightarrow 2^{AP}$. Let the kth element of σ (starting with $k = 0$) be denoted by $\sigma[k]$.

We denote by $(\sigma, k, \alpha) \models \psi$, the fact that trace σ satisfies the formula ψ at position k with valuation α. The *satisfaction relation* $\models$ is defined recursively as follows.
For atomic propositions and Boolean connectives:

– $(\sigma, k, \alpha) \models p$ iff $p \in \sigma[k]$; $(\sigma, k, \alpha) \models \neg\psi$ iff $(\sigma, k, \alpha) \not\models \psi$
– $(\sigma, k, \alpha) \models (\psi_1 \wedge \psi_2)$ iff $(\sigma, k, \alpha) \models \psi_1$ and $(\sigma, k, \alpha) \models \psi_2$.
– $(\sigma, k, \alpha) \models (\psi_1 \vee \psi_2)$ if $(\sigma, k, \alpha) \models \psi_1$ or $(\sigma, k, \alpha) \models \psi_2$.

For the LTL operators:

– $(\sigma, k, \alpha) \models \bigcirc\psi$ iff $|\sigma| > k + 1$ and $(\sigma, k + 1, \alpha) \models \psi$.
– $(\sigma, k, \alpha) \models (\psi_1 \,\mathcal{U} \psi_2)$ if there exists i, $k < i < |\sigma|$ where $(\sigma, i, \alpha) \models \psi_2$, and for each j, $k \leq j < i$, $(\sigma, j, \alpha) \models \psi_1$.
– $(\sigma, k, \alpha) \models (\psi_1 \mathcal{R} \psi_2)$ if for each $k \leq i < |\sigma|$, it holds that either $(\sigma, i, \alpha) \models \psi_2$ or there exists j, $k \leq j < i$ such that $(\sigma, k + j, \alpha) \models \psi_1$.

We use the following standard abbreviations: $\Diamond\varphi = (\mathit{true}\, \mathcal{U} \varphi)$, $\Box\varphi = (\mathit{false}\, \mathcal{R} \varphi)$. The parametric operators are defined as follows:

– $(\sigma, k, \alpha) \models \Diamond_{\leq x}\psi$ if there exists $0 \leq i \leq \alpha(x)$, where $k + i < |\sigma|$, such that $(\sigma, k + i, \alpha) \models \psi$;
– $(\sigma, k, \alpha) \models \Box_{\leq x}\psi$ if for all $0 \leq i \leq \alpha(x)$, where $k + i < |\sigma|$, $(\sigma, k + i, \alpha) \models \psi$;

We also write $(\sigma,\alpha) \models \varphi$ for $(\sigma,0,\alpha) \models \varphi$. We can extend our syntax and semantic definitions to allow also constants in addition to (or instead of) the variable parameters: Constants are simply parameters that have the same values under each valuation.

The semantics of the derived parametric operators is given by the following equalities (see [2], Lemma 2.2):

- $\Diamond_{>x}\psi = \Box_{\leq x}\Diamond\bigcirc\psi$;
- $\Box_{>x}\psi = \Diamond_{\leq x}\Box\bigcirc\psi$;
- $\psi_1\,\mathcal{U}_{\leq k}\,\psi_2 = ((\psi_1\,\mathcal{U}\,\psi_2) \wedge \Diamond_{\leq k}\psi_2)$;
- $\psi_1\,\mathcal{R}_{\leq k}\,\psi_2 = ((\psi_1\,\mathcal{R}\,\psi_2) \vee \Box_{\leq k}\psi_2)$;
- $\psi_1\,\mathcal{U}_{>k}\,\psi_2 = \Box_{\leq k}(\psi_1 \wedge \bigcirc(\psi_1\,\mathcal{U}\,\psi_2))$;
- $\psi_1\,\mathcal{R}_{>k}\,\psi_2 = \Diamond_{\leq k}(\psi_1 \vee \bigcirc(\psi_1\,\mathcal{R}\,\psi_2))$.

Each of the parametric operators is either upward or downward closed. $\Diamond_{\leq x}$ is *upward closed*: if $\Diamond_{\leq x}$ for some value $\alpha(x) = a$, then $\Diamond_{\leq x}$ also holds for any $\alpha(x) = b$ with $b > a$. If an operator is upward closed, we want to *minimize* the value we report. However, as this operator can hold in multiple suffixes of the measured sequence, we need to report on a value that would guarantee all of them, hence the *maximum* among these minimal values. Likewise, $\Box_{\leq x}$ is *downward closed*: if $\Box_{\leq x}$ for some value $\alpha(x) = a$, then $\Box_{\leq x}$ also holds for any $\alpha(x) = a$ with $0 \leq b < a$. If an operator is downward closed, we want to *maximize* the value we report. However, as this operator can hold in multiple suffixes of the measured sequence, we need to report on a value that would guarantee all of them, hence the *minimum* among these maximal values.

We assume that the PLTL formulas are in negation normal form (i.e., negations may only occur in front of atomic propositions). Negation normal form can be established by pushing negations inward according to the usual rewrite rules for LTL, e.g., $\neg(\psi_1\,\mathcal{U}\,\psi_2) = (\neg\psi_1\,\mathcal{R}\,\neg\psi_2)$, and, additionally, the following equivalence for the parametric operators: $\neg\Box_{\leq x}\psi = \Diamond_{\leq x}\neg\psi$. This transformation increases the size of the formula only by a constant factor. We also assume that the parameterized operators are only $\Box_{\leq x}$ and $\Diamond_{\leq x}$. The transformation according to the equalities for the derived parametric operators can result in an exponential explosion in the size of the formula. However, one does not need to explicitly represent such a formula: one can introduce "formula variables" to name repeating subformulas and use them repeatedly. Indeed, in all algorithms in this paper, one does not pay for the repetition of subformulas resulting from the rewriting [4].

Unique Measures. An attractive feature of PLTL is that the logic permits more than one parameter in the same formula. As discussed in the introduction, this means, however, that a trace can satisfy the formula with multiple incomparable value assignments: for example, the formula $\Diamond_{\leq x}\Diamond_{\leq y}p$ is satisfied on a trace where p is false in the first position and true in the second position both for the value assignment $\alpha : x \mapsto 1, y \mapsto 0$ and for the value assignment $\alpha' : x \mapsto 0, y \mapsto 1$.

To avoid such ambiguities, we introduce a total (i.e., linear) *priority* order $\gg$ on the parameters in X. Let $\max(X)$ be the maximum element of X according to $\gg$. The priority order induces a total order $\sqsupset$ on value assignments where $\alpha_1 \sqsupset \alpha_2$ if, for $x = \max(X)$,

- $(\alpha_1(x) - \alpha_2(x)) > 0$ and we *maximize* x (i.e., the operator of x is downward closed) or
- $(\alpha_1(x) - \alpha_2(x)) < 0$ and we *minimize* x (i.e., the operator of x is upward closed) or
- $\alpha_1(x) = \alpha_2(x)$ and $\alpha_1 \setminus x \sqsupseteq \alpha_2 \setminus x$.

The *measure* of a PLTL formula φ over an infinite trace σ is the optimal (with respect to $\sqsupseteq$) value assignment α such that $(\sigma, 0, \alpha) \models \varphi$.

3 Offline Measuring

We present a first algorithm for measuring a given finite trace. We call the algorithm *offline*, because it requires access to the trace positions in reverse chronological order; this type of access is possible if the trace has been stored before its analysis. The algorithm is less appropriate for the online setting of monitoring, where the trace becomes available one position at a time. We will study online measuring in Sections 4 and 5.

Intuitively, we focus first on the parameter with highest priority, setting up the other variables to a default value of 0 for a maximizing parameter, and $|\sigma| + 1$ for a minimizing parameter. We perform a binary search on the value of this variable, hence are left with a formula with constant parameters. After finding the optimal (minimal or maximal) value for this parameter in this way, we fix it, and move to optimize the next highest priority parameter and so forth.

Checking Formulas with Constant Parameters. We begin with the simple case of PLTL formulas *without* parameters that may still contain parametric operators that refer to constants. In this case, we are only interested in the truth value, not in an actual measurement.

Let σ be a finite trace and let φ be a PLTL formula in normal form with constants instead of variable parameters. We check the satisfaction of φ in a backward traversal of σ. During the traversal, we maintain for every subformula ψ the truth value b_ψ, which indicates whether ψ is satisfied on the suffix from the currently considered position. For every subformula ψ that starts with a parametric operator (referring to some constant k) we additionally maintain a counter c_ψ, which indicates for $\psi = \Diamond_{\leq k}\mu$ the number of steps until μ is satisfied, and for $\psi = \Box_{\leq k}\mu$ the number of steps until μ is falsified, respectively; in case μ (respectively, $\neg\mu$) never become true, we set $c_\psi = \bot$.

Before we process the last event in the trace we set up the values as follows:

- $b_{\Diamond_{\leq k}\psi} = \mathit{false}$, $c_{\Diamond_{\leq k}\psi} = \bot$,
- $b_{\Box_{\leq k}\psi} = \mathit{true}$, $c_{\Box_{\leq k}\psi} = \bot$,
- $b_{\bigcirc\psi} = \mathit{false}$.
- $b_{(\psi_1 \mathcal{U} \psi_2)} = \mathit{false}$.
- $b_{(\psi_1 \mathcal{R} \psi_2)} = \mathit{true}$.

For the Boolean connectives and the non-parametric operators, the backward propagation proceeds as in a standard backward update algorithm for LTL (c.f., [6]). We denote the values for the current level i with b_ψ and c_ψ and the values for the previously considered level $i+1$ with b'_ψ and c'_ψ. For Boolean combinations we evaluate bottom-up as follows: $b_{(\varphi\wedge\psi)} = (b_\varphi \wedge b_\psi)$ and $b_{(\varphi\vee\psi)} = (b_\varphi \vee b_\psi)$.

For the non-parametric temporal operators, we propagate the truth values as follows:

– $b_{\bigcirc\psi} = b'_{\psi}$
– $b_{\psi_1 \vee \psi_2} = b_{\psi_1} \vee b_{\psi_2}$
– $b_{(\psi_1 \mathcal{U} \psi_2)} = (b_{\psi_2} \vee (b_{\psi_1} \wedge b'_{\psi_1 \mathcal{U} \psi_2}))$
– $b_{(\psi_1 \mathcal{R} \psi_2)} = (b_{\psi_2} \wedge (b_{\psi_1} \vee b'_{\psi_1 \mathcal{R} \psi_2}))$

For the parametric operators, we need to update the counters. In the following we assume $\perp + 1 = 0$ and $\perp \not\leq i$ for any $i \in \mathbb{N}$.

– $c_{\diamond_{\leq k}\psi}$:= if b_ψ then 0
  else if $c'_{\diamond_{\leq k}\psi} < k$ then $c'_{\diamond_{\leq k}\psi} + 1$ else $\perp$.

– $c_{\square_{\leq k}\psi}$:= if $\neg b_\psi$ then $\perp$
  else if $c'_{\square_{\leq k}\psi} < k$ then $c'_{\square_{\leq k}\psi} + 1$ else 0.

The truth values of the parametric operators can then be derived from the counter values: $b_{\diamond_{\leq k}\psi} = (c_{\diamond_{\leq k}\psi} \leq k)$; $b_{\square_{\leq k}\psi} = (c_{\square_{\leq k}\psi} \geq k)$.

In order to check φ on σ, we thus update two values for every subformula and trace position. The running time of our algorithm is therefore in $O(2^{|\varphi|} \times |\sigma|)$.

Measuring Formulas with at least one Parameter. In the case that φ contains a single parameter x, we know that the possible values of x are bounded by the length of the trace. We carry out a binary search to find the best value. The running time of the algorithm thus increases by a logarithmic factor in the length of the trace: $O(2^{|\varphi|} \times |\sigma| \times log|\sigma|)$. In case that φ contains $n > 1$ parameters, we focus on the parameter x with the highest priority by replacing all other parameters by their 'weakest' values, i.e., 0 if we maximize and $|\sigma| + 1$ if we minimize that parameter. This clearly does not affect the value of x in the measure. Once the value of x is obtained, we replace x with its value, and continue with the parameter with the next-highest priority.

The algorithm from the single-parameter case is therefore applied n times, where n is bounded by $|\varphi|$. We therefore obtain the following complexities.

Theorem 1. *Let φ be a PLTL formula and σ be a finite trace. With direct access to all trace positions, the measure of φ on σ can be computed in space $O(|\varphi| \times |\sigma| \times \log|\sigma|)$ and time $O(|\varphi| \times 2^{|\varphi|} \times |\sigma| \times \log|\sigma|)$.*

4 Online Measuring is Hard

The algorithms from the previous section assume direct access to the full trace. Since huge traces are common in practice for runtime verification, in fact, their size may not be a priori bounded, one would like to avoid storing the full trace, and instead only work with a logarithmic representation, such as the current value of a fixed number of counters. The following theorem shows that, unfortunately, such a monitoring algorithm cannot exist.

Theorem 2. *There is no online measuring algorithm for PLTL that uses only logarithmic space in the length of the trace.*

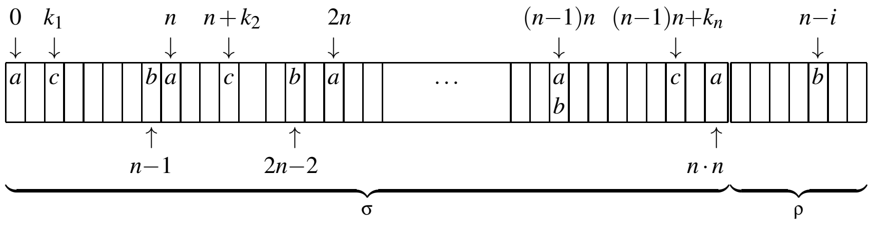


Fig. 1. Sequences σ and ρ in the proof of Theorem 2

Proof. Suppose that such an online algorithm exists. We run the algorithm on the formula $\Box(a \rightarrow ((\Diamond_{\leq x} b) \vee (\Diamond_{\leq y} c)))$, where $x \gg y$. We will show that there is a sequence σ of length $O(n^2)$, such that the memory of the monitor must, after processing σ, contain all elements of an arbitrary chosen set $K = \{k_1, k_2, \ldots, k_n\}$ of n natural numbers.

Assume, without loss of generality, that $k_1 < k_2 < \ldots < k_n$. The sequence σ is constructed as follows. There is an a in the first position and then again after n steps, after $2n$ steps, and so on, for a total of $n+1$ times. In the first interval between two occurrences of a, there is a b after $n-1$ steps following the first a, and a c after k_1 steps following the first a, in the second interval, there is a b after $n-2$ steps following the first a and a c after k_2 steps, and so on. The construction of σ is illustrated on the left in Figure 1.

The monitor must keep the entire set K in memory after processing σ, because we can force the monitor to retrieve $k_i \in K$ for any $i = 1, \ldots n$, by extending σ with a suitable sequence ρ such that the y-measurement of $\sigma \cdot \rho$ is k_i. The extension, after the last a, consists of another $n-1$ steps with a b in the $(n-i)$th step and no further or c (or a). The construction is illustrated on the right in Figure 1. With only logarithmic memory, it is impossible to store K. Logarithmic space can only distinguish $O(n)$ cases; however, there can be 2^n different sets K.

The measurement of the higher priority variable x in $\sigma \cdot \rho$ cannot be smaller than $x = n - i + 1$, since the last a is followed by a b after that distance, but not with any c. For the purposes of measuring y, we only need to consider the first $i-1$ occurrences of a, because for all other occurrences, the left disjunct, with the chosen measure of x being at least $x = n - i + 1$, is always satisfied. In order not to increase the value of x beyond $n - i + 1$, we need to satisfy the disjunction for each of the first $i-1$ occurrences of a (which follow by a b at a larger distance, namely, $n-i+2, n-i+3 \ldots n$) using its righthand side, i.e., through the first occurrence of a c after each a. In order to guarantee that all these distances from a to the first subsequent c are satisfying the formula with the measurement of y, we must choose y as the maximal value of them. As the distances appear in ascending order, we must have $y = k_{i-1}$. □

5 Online Measuring in Logarithmic Space

In this section, we present online measuring algorithms that only need logarithmic space in the length of the trace. Since we know from Theorem 2 that disjunctions between subformulas with parameters make this impossible for PLTL, we must look at syntactic or semantic variations of PLTL that "determinize" such disjunctions. We start with a syntactic fragment based on deterministic LTL [11]: in deterministic LTL, the only allowed disjunctions are of the form $((p \wedge \psi_1) \vee (\neg p \wedge \psi_2))$, where the subformulas ψ_1 and ψ_2 are guarded by a proposition p; since the value of p is immediately available, the choice of the disjunct is deterministic. Indeed, as we show in Section 5.1, runtime measuring of deterministic PLTL can be done with logarithmic cost in the length of the trace. Deterministic PLTL is, however, not completely satisfying as a logic for runtime measuring, because it is less expressive than full LTL: deterministic LTL can express exactly the properties in the intersection of LTL and ACTL [11].

We solve this problem in Section 5.2 by introducing *unambiguous* PLTL, which maintains the full expressiveness of LTL. Instead of syntactically restricting the possible disjunctions, we only *disambiguate* the PLTL semantics with respect to the measuring. For example, in a disjunction $(\psi_1 \vee \psi_2)$, we *only* measure ψ_2 if ψ_1 is false for all possible instantiations of the parameters. Again, the complexity of online measuring drops from linear to logarithmic in the length of the trace.

5.1 Deterministic PLTL

We define deterministic PLTL in analogy to deterministic LTL [11] by restricting the syntax of PLTL such that disjunctions and eventualities are always guarded by atomic propositions.

Syntax and Semantics. The *syntax* of PLTL$^{\text{det}}$ is given, for a set of atomic propositions AP and a set of parameter variables V as follows:

$$\psi ::= \mathit{true} \mid p \mid \neg\psi \mid \psi \wedge \psi \mid (p \wedge \psi) \vee (\neg p \wedge \psi) \mid \bigcirc\psi \mid \Diamond p \mid (p \wedge \psi)\ \mathcal{U}\ (\neg p \wedge \psi) \mid \\ \Box_{\leq x} p \mid \Diamond_{\leq x} p$$

PLTL$^{\text{det}}$ is a sublogic of PLTL; the semantics remains the same.

Measuring Automata. We construct a monitor in the form of an extended finite-state automaton, which maintains the current measurements in a fixed number of integer variables. We begin with a formal definition of measuring automata.

A measuring automaton is a deterministic finite-state automaton extended with a set of variables, which are used to store data needed to compute the measure. The variables are initialized with either 0 or ∞. In each step, the automaton may update the integer variables with a reset to 0, an increment by 1, or by computing the minimum or maximum of two values. When the automaton reaches a final state it accepts the input word and outputs its measurement based on the state and the values of the integer variables.

Definition 1. *A* measuring automaton $(\Sigma, \Omega, Q, q_0, X, \theta, \delta, \gamma, F, \omega)$ *consists of an input alphabet* Σ*, an output domain* Ω*, a finite set of states* Q*, an initial state* q_0*, a finite set of variables* X*, an initial assignment* $\theta : X \to \{0, \infty\}$*, a transition function* $\delta : Q \times \Sigma \to (Q \cup \{\bot\})$*, an update function* $\gamma : Q \times \Sigma \to (X \to \mathbb{N}) \to (X \to \mathbb{N})$*, a set of final states* F*, and an output function* $\omega : F \times (X \to \mathbb{N}) \to \Omega$*. The update function* γ *is restricted to one* *of the following operations for each variable $x \in X$:* reset $x := 0$, *an* increment $x := y+1$, *or with the* maximum *or* minimum *of two values:* $x := \min(x,y)$ *or* $x := \max(x,y)$, *where* y *is some other variable* $y \in X$.

A *run* of a measuring automaton $\mathcal{A} = (\Sigma, \Omega, Q, q_0, X, \theta, \delta, \gamma, F, \omega)$ on an input sequence $\sigma = \sigma_0\sigma_1 \dots \sigma_n \in \Sigma^*$ is a sequence $(s_0, \eta_0)(s_1, \eta_1)\dots(s_n, \eta_n)$ of configurations, where each configuration is a pair (s_i, η_i) of a state s_i and a valuation $\eta_i : X \to \mathbb{N}$ of the integer variables, such that

- $s_0 = q_0$
- $\eta_0 = \theta$
- $s_{i+1} = \delta(s_i, \sigma_i)$ for $i = 0 \dots (n-1)$
- $\eta_{i+1} = \gamma(s_i, \sigma_i)(\eta_i)$ for $i = 0 \dots (n-1)$
- $s_n \in F$.

The *result* of the run is $\omega(s_n, \eta_n)$. For every input sequence, $\mathcal{A}$ has either no run at all or a unique run. If $\mathcal{A}$ has a run on σ, we say that $\mathcal{A}$ *accepts* σ with result $\omega(s_n, \eta_n)$.

Since the only allowed update operations are reset, increment, maximum, and minimum, the values of the variables are always either ∞ or bounded by the length of the input sequence. These values can therefore be represented in logarithmic space in the length of the input.

Lemma 1. *The configuration of a measuring automaton can be represented in logarithmic space in the length of the input sequence.*

From Formulas to Automata. We measure formulas of PLTL$^{\text{det}}$ with a measuring automaton with input alphabet $\Sigma = 2^{AP}$ and output domain $\Omega : V \to \mathbb{N}$, consisting of the evaluations of the parameters. The state space of the automaton is based, as in classic LTL-to-automata translations, on the *closure* of the formula.

Definition 2. *The* closure φ, *denoted by* $cl(\varphi)$, *of a PLTL formula* φ *is the set of PLTL formulas that includes all the subformulas of* φ *and the negations of the non-parametric subformulas of* φ.

The states of the measuring automaton consist (in addition to a unique initial state q_0) of subsets of the closure called *atoms*. Intuitively, an atom represents the state of the temporal specification after processing a prefix of the trace.

Definition 3. *An* atom *of a PLTL formula* φ *is subset of formulas from* $cl(\varphi)$ *that is consistent with respect to propositional logic, locally consistent with respect to the until, release, globally, and parametric globally operators, and maximal.*

- *A subset* $A \subseteq cl(\varphi)$ *of the closure is* consistent with respect to propositional logic *if the following conditions hold:* $(\psi_1 \wedge \psi_2) \in A$ *iff* $\psi_1 \in A$ *and* $\psi_2 \in A$; $\psi \in A$ *implies that* $\neg\psi \notin A$; *and true* $\in cl(\varphi)$ *implies that true* $\in A$.
- *A subset* $A \subseteq cl(\varphi)$ *of the closure is* locally consistent with respect to the until operator *if for all* $(\psi_1 \, \mathcal{U} \, \psi_2) \in cl(\varphi)$ *the following conditions hold:* $\psi_2 \in A$ *implies that* $(\psi_1 \, \mathcal{U} \, \psi_2) \in A$; *and* $(\psi_1 \, \mathcal{U} \, \psi_2) \in A$ *and* $\psi_2 \notin A$ *implies that* $\psi_1 \in A$;

- *A subset $A \subseteq cl(\varphi)$ of the closure is* locally consistent with respect to the release operator *if for all $(\psi_1 \mathcal{R} \psi_2) \in cl(\varphi)$ the following conditions hold: $\psi_1 \in A$ and $\psi_2 \in A$ implies that $(\psi_1 \mathcal{R} \psi_2) \in A$; and $(\psi_1 \mathcal{R} \psi_2) \in A$ implies that $\psi_2 \in A$;*
- *A subset $A \subseteq cl(\varphi)$ of the closure is* locally consistent with respect to the globally operator *if for all $\Box\psi \in cl(\varphi)$ it holds that if $\Box\psi \in A$ then $\psi \in A$;*
- *A subset $A \subseteq cl(\varphi)$ of the closure is* locally consistent with respect to the parametric globally operator *if for all $\Box_{\leq x}\psi \in cl(\varphi)$ it holds that if $\Box_{\leq x}\psi \in A$ then $\psi \in A$.*
- *A subset $A \subseteq cl(\varphi)$ of the closure is* maximal *if, for all non-parametric subformulas $\psi \in cl(\varphi)$, we have that either $\psi \in A$ or $\neg\psi \in A$.*

Let At_φ denote the set of atoms of φ. We consider the following successor relation on atoms:

Definition 4. *Let $\rightarrow \subseteq At_\varphi \times 2^{AP} \times At_\varphi$ be a* successor relation *between atoms of φ and for each t, $e \subseteq AP$, t', we have $t \xrightarrow{e} t'$ if t' is the smallest set s.t.*

- *$t' \cap AP = e$.*
- *If $\bigcirc\psi \in t$ then $\psi \in t'$.*
- *If $\Box\psi \in t$, then $\Box\psi \in t'$.*
- *If $(\psi_1 \mathcal{U} \psi_2) \in t$ then either $\psi_2 \in t$ or $\psi_1 \in t$ and $(\psi_1 \mathcal{U} \psi_2) \in t'$.*
- *If $\Diamond\psi \in t$ then $\psi \in t$ or $\Diamond\psi \in t'$.*
- *If $\Diamond_{\leq x}\psi \in t$ then $\psi \in t$ or $\Diamond_{\leq x}\psi \in t'$.*
- *If $\Box_{\leq x}\psi \in t$ then $\psi \in t$ and $\psi \notin t'$ or $\Box_{\leq x}\psi \in t'$.*

For PLTL$^{\text{det}}$, this successor relation leads to a deterministic automaton.

Lemma 2. *For every atom $t \in At(\varphi)$ of a PLTL^{det} formula φ and every event e there is at most one atom $t' \in At(\varphi)$ such that $t \xrightarrow{e} t'$.*

The transition function δ of the measuring automaton is therefore directly based on the successor relation:

- For the initial state q_0, the successor $\delta(q_0, e)$ is the unique atom that contains φ and is consistent with e, i.e., the atom $t \in At_\varphi$ with $\varphi \in t$ and $t \cap AP = e$, or $\bot$ if no such atom exists.
- For every other state $t \in At_\varphi$, the successor $\delta(t, e)$ is, whenever it exists, the unique atom t' with $t \xrightarrow{e} t'$. If no successor atom exists, then $\delta(t, e) = \bot$.

The set X of variables of the measuring automaton contains two variables n_v and m_v for each parameter $v \in V$. The variable n_v is a counter, similar to the counter used in the offline algorithm: n_v indicates the number of steps since the formula that starts with the operator that is parametric in v has existed in the atom. Since the same formula may be generated several times along the trace, e.g., the subformula $\Diamond_{\leq x}q$ of the formula $\Box(p \rightarrow \Diamond_{\leq x}q)$ (written in negation normal form) is generated whenever p is true, we additionally keep track of the worst (maximal, if we need to minimize, or minimal, if we need to maximize the measure) value of n_v seen so far (thus ensuring that the final measurement will cover all occurrences). This is the purpose of the second variable m_v:

for the upward-closed operator $\Diamond_{\leq x}$, m_x contains the greatest value of n_x seen so far, for the downward-closed operator $\Box_{\leq x}$, m_x contains the smallest value of n_x seen so far. In θ, we initialize n_x with 0 and m_x with ∞ for $\Box_{\leq x}$ and with 0 for $\Diamond_{\leq x}$.

For the update function $\gamma(s,e)$, which maps a state and an input to a mapping between successive valuations, we need to distinguish situations where a parametric formula is freshly generated from situations where the formula is present in order to continue a measurement started earlier.

Definition 5. *A formula $\psi \in t$ is* generated *in a pair of atoms $t, t' \in At_\varphi$, denoted by generated(ψ, t, t'), if one of the following conditions is true:*

- *ψ is the direct subformula of some other formula in t';*
- *$\bigcirc\psi$ is a formula in t.*

For simplicity of notation, we extend *generated*(ψ, s, s') to pairs of states $s, s' \in Q$ of the measuring automaton, where we set *generated*$(\psi, q_0, t') = \mathit{true}$ for all $t' \in At_\varphi, \psi \in t'$ to also cover the designated initial state. The updates of m_x and n_x take into account whether we want to maximize or minimize a subformula. Also, the updates take into account the fact that a measured subformula may be regenerated while predecessor atoms already include that subformula, hence it is already in the process of being measured. For example, when monitoring $\Box(p \to \Diamond_{\leq x} q)$, the subformula $\Diamond_{\leq x} q$ may be generated, or a result of it appearing in the previous atom, or both. We need to measure an overall worst *minimal* value (i.e., maximum among minimal values from the time $\Diamond_{\leq x} p$ is generated in an atom until p it holds a subsequent atom). In this case, we ignore any new generation of the subformula in the current atom. The situation is reversed when we have an $\Box_{\leq x} q$ subformula: Since we need the worst among *maximal* measurements, we ignore the occurrence of the subformula that is propagated from a predecessor atom in favor of a current generation, and start to count from fresh.

```
For ◇_{≤x}η ∈ δ(s,e): if η ∈ t′ then m_x := max(m_x, n_x); n_x := 0
                      else m_x := m_x; n_x := n_x + 1.

For □_{≤x}η ∈ δ(s,e): if η ∈ δ(s,e) then if generated(□_{≤x}η, s, δ(s,e))
                                           then m_x := m_x; n_x := 0
                                           else m_x := m_x; n_x := n_x + 1;
                      else m_x := min(m_x, n_x); n_x := 0.
```

If x does not occur in the new atom, then m_x and n_x remain unchanged: $m_x := m_x$ and $n_x := n_x$. The final states of the measuring automaton are the atoms without unfulfilled obligations, i.e., $t \in F$, iff for all $\eta\, \mathcal{U} \mu \in t$ also $\mu \in t$, for all $\Diamond_{\leq x}\mu$ also $\mu \in t$, and there is no formula $\bigcirc\psi$ in t. For such states, ω reports the parameter assignment $v \mapsto m_v$ for all parameters, with the exception of remaining formulas of type $\Box_{\leq x}\mu$, where it reports $\min(m_x, n_x)$ to take care of a new possible minimum on the last event on the trace.

Theorem 3. *For every PLTL^{det} formula φ there exists a measuring automaton $\mathcal{A}_\varphi = (\Sigma, \Omega, Q, q_0, X, \theta, \delta, \gamma, F, \omega)$ with a linear number of states Q in $|\varphi|$ and a linear number of variables X in $|\varphi|$ such that for every sequence $\sigma \in (2^{AP})^*$, σ is accepted by $\mathcal{A}_\varphi$ with result r iff r is the measure of φ on σ.*

Soundness. We split the correctness argument of Theorem 3 into two lemmata:

Lemma 3. *If there exists a run of $\mathcal{A}_\varphi$ on the trace $\sigma = e_0 e_1 \dots e_{n-1} \in (2^{AP})^*$ with result r, then $(\sigma, 0, r) \models \varphi$.*

Lemma 4. *For a trace $\sigma = e_0 e_1 \dots e_{n-1} \in (2^{AP})^*$, if $(\sigma, 0, r) \models \varphi$, then there exists a run $\pi = (t_0, \eta_0), (t_1, \eta_1) \dots (t_n, \eta_n)$ of $\mathcal{A}_\varphi$ on σ with result r', where $r' \sqsubseteq r$.*

Because $\mathcal{A}_\varphi$ has at most one run on a given trace and therefore a unique result, provided that some run exists, Lemmata 3 and 4 imply that the result of $\mathcal{A}_\varphi$ is the measure of φ on the given trace. To prove Lemma 3, we first establish that the formulas in the atoms are satisfied for the respective suffixes of the trace.

Lemma 5. *If there exists a run of φ in $\mathcal{A}_\varphi$ with atom sequence $t_0 \dots t_n$, then we have that for all subformulas ψ, for all positions i, if $\psi \in t_i$ then $(\sigma, i-1, r) \models \psi$.*

The proof of Lemma 5 is by induction on the length of the trace, progressing backwards from the last position. It remains to show that the result of the run satisfies φ.

Lemma 6. *If there exists a run $\pi = (t_0, \eta_0), (t_1, \eta_1) \dots (t_n, \eta_n)$ in $\mathcal{A}_\varphi$ with result r, we have that $(\sigma, 0, r) \models \varphi$.*

To prove Lemma 6, we show, inductively, that for a subformula $\Diamond_{\leq x}\psi$ in atom t_i, the distance to the next atom with $[\psi]$ is at most $r(x) - \eta_i(n_x)$ steps; and, likewise, that for a subformula $\Box_{\leq x}\psi$ in atom t_i, the distance to the next atom with $\neg[\psi]$ is at least $r(x) - \eta_i(n_x)$ steps.

For the reverse direction, stated as Lemma 4, we first construct the sequence of atoms corresponding to the given trace. Based on the semantics definition, we show inductively that the subformulas in the atoms hold over the respective suffices.

Lemma 7. *For a trace $\sigma = e_0 e_1 \dots e_{n-1}$ and a deterministic PLTL formula φ, if $(\sigma, 0, r) \models \varphi$, then there exists an atom sequence $t_0 \dots t_n$ such that t_0 is the unique atom that contains φ and is consistent with e_0, $t_i \xrightarrow{s_i} t_{i+1}$ for all $i = 0 \dots n-2$, and for every subformulas ψ and position $i = 0 \dots n-1$, if $\psi \in t_i$, $(\sigma, i-1, r) \models \psi$.*

We complete the atom sequence of Lemma 7 into a complete run by computing the values of n_x and m_x for each parameter x and each trace position according to the definition of the automaton.

Lemma 8. *For a trace $\sigma = e_0 e_1, \dots e_{n-1}$ and a deterministic PLTL formula φ, if $(\sigma, 0, r) \models \varphi$, then there exists a run of $\mathcal{A}_\varphi$, $\pi = (t_0, \eta_0), (t_1, \eta_1) \dots (t_n, \eta_n)$ where $t_i \xrightarrow{e_i} t_{i+1}$ and for all subformulas and positions i, if $\psi \in t_i$, $(\sigma, i-1, r) \models \psi$, with result $\omega(e_n, \eta_n) \sqsubseteq r$.*

To prove the claim in Lemma 8, that the result of the run is at least as good as r, we show, inductively, that for each subformula $\Diamond_{\leq x}\psi$ and each trace position i, $\eta_i(n_x)$ is less than or equal to the difference of $r(x)$ and the distance of the closest atom that contains $[\psi]$; and that for each subformula $\Box_{\leq x}\psi$ and each trace position i, the sum of $\eta_i(n_x)$ and the distance of the closest atom that contains $\neg[\psi]$ is greater than or equal to

$r(x)$. Since m_x maintains for $\Diamond_{\leq x}\psi$ and for $\Box_{\leq x}\psi$, the maximum and minimum measure, respectively, the claim of Lemma 8 follows.

From Theorem 3 and Lemma 1 it follows that the space required by the online monitor is linear in the size of the specification and logarithmic in the length of the trace.

Theorem 4. *A PLTL^{det} formula φ can be measured in linear space in the size of φ and logarithmic space in the length of the trace.*

5.2 Unambiguous PLTL

As discussed in the introduction of Section 5, the disadvantage of the syntactic restriction in deterministic PLTL is that it affects the expressiveness of the logic. In *unambiguous PLTL*, we modify the semantics of PLTL, rather than its syntax, in order to determinize the measuring behavior. Because the change does not affect the truth value of the non-parametric subformulas, we maintain the full expressiveness of LTL. In particular, under the unambiguous interpretation we give priority in $(\varphi \vee \psi)$ to φ, and measure according to ψ only when $[\varphi]$ does not hold. Similarly, for $\Diamond_{\leq x}\varphi$, we measure x to the *first* occurrence where φ holds.

Syntax and Semantics. We use the full PLTL syntax as defined in Section 2. The changes in the semantics (we do not redefine the cases that remain the same) are as follows:

- $(\rho,k,\alpha) \models (\psi \vee \eta)$ if $(\rho,k,\alpha) \models \psi$ or $(\rho,k,\alpha) \models (\neg[\psi] \wedge \eta)$;
- $(\rho,k,\alpha) \models (\psi\, \mathcal{U} \eta)$ if there exists i where $|\sigma| < k+i$, such that $(\rho,k+i,\alpha) \models \eta$, and for each j, $0 \leq j < i$, $(\rho,k+j,\alpha) \models \psi \wedge [\neg\eta]$;
- $(\sigma,k,\alpha) \models \Diamond_{\leq x}\eta$ if there exists $0 \leq i \leq \alpha(x)$, where $k+i < |\sigma|$, such that $(\sigma,k+i,\alpha) \models \eta$ and for each j, $0 \leq j < i$, $(\rho,k+j,\alpha) \models [\neg\eta]$;
- $(\sigma,k,\alpha) \models (\psi\, \mathcal{R} \eta)$ if for each $k \leq i < |\sigma|$, either $(\sigma,i,\alpha) \models \eta \wedge [\neg\psi]$ or there exists j, $k \leq j < i$ such that $(\sigma,k+j,\alpha) \models \psi$.

The definition uses the LTL abstraction $[\psi]$ of a PLTL formula ψ, which is the LTL formula that is satisfied exactly if the PLTL formula is satisfiable for some instance of the parameters, i.e.,

- $[\Diamond_{\leq x}\psi] = \Diamond[\psi]$
- $[\Box_{\leq x}\psi] = [\psi]$
- $[p] = p; [\neg p] = \neg p; [(\psi_1 \wedge \psi_2)] = ([\psi_1] \wedge [\psi_2]); [(\psi_1 \vee \psi_2)] = ([\psi_1] \vee [\psi_2])$;
- $[\bigcirc\psi] = \bigcirc[\psi]; [\Diamond\psi] = \Diamond[\psi]; [\Box\psi] = \Box[\psi]$;
- $[(\psi_1\, \mathcal{U} \psi_2)] = ([\psi_1]\, \mathcal{U} [\psi_2]); [(\psi_1\, \mathcal{R} \psi_2)] = ([\psi_1]\, \mathcal{R} [\psi_2])$.

Example. Consider the PLTL formula $\varphi = a\, \mathcal{U} \Diamond_{\leq x} b$ on the trace $\sigma = \{a\}\{a,b\}\emptyset$. According to the standard semantics from Section 2, we have both $(\sigma,0,x \mapsto 1) \models \varphi$, because $(\sigma,0,x \mapsto 1) \models \Diamond_{\leq x}b$, and $(\sigma,0,x \mapsto 0) \models \varphi$ because $(\sigma,0,x \mapsto 0) \models a$ and $(\sigma,1,x \mapsto 0) \models \Diamond_{\leq x}b$. The result according to the standard semantics is therefore 0. According to the unambiguous semantics, we only have $(\sigma,0,x \mapsto 1) \models \varphi$, because $(\sigma,0,x \mapsto 1) \models [\Diamond_{\leq x}b] = \Diamond b$, and $(\sigma,0,x \mapsto 1) \models \Diamond_{\leq x}b$. The result is therefore 1.

Note that for formulas in the syntax of deterministic PLTL, the unambiguous and the standard semantics agree. Unambiguous PLTL is thus a strict generalization of deterministic PLTL.

From Formulas to Automata. Measuring unambiguous PLTL is more difficult than deterministic PLTL, since the disjuncts are no longer guarded by atomic propositions, we generally do not know the truth value of a disjunct until the end of the trace. Our construction exploits the fact that the counting that is needed to compute the measure of the trace is deterministic in the truth value of the temporal subformulas. While the actual value of the future formulas is not known during monitoring, it is possible to split the analysis into a fixed set of cases based on the possible truth values of the temporal subformulas.

The states of the measuring automaton are *sets* of atoms. The intuitive idea is that each atom represents a possible future behavior. The sets of atoms correspond to a determinization of the possible futures. As the execution unrolls, some of the future possibilities are ruled out. As before, an atom is a subset of the formulas of the closure. We extend the closure of the PLTL formula φ with the subformulas of $[\varphi]$ and the negations of the subformulas of $[\varphi]$. Under the unambiguous semantics, we additionally require unambiguity with respect to disjunction and until:

Definition 6. *An* atom *of an unambiguous PLTL formula φ is subset of formulas from $cl(\varphi)$ that is consistent with respect to propositional logic, locally consistent with respect to the until, release, globally, and parametric globally operators, maximal, and unambiguous with respect to disjunction and until.*

– *A subset $A \subseteq cl(\varphi)$ of the closure is* unambiguous with respect to disjunction *if, for every $\eta \vee \mu \in A$ either $[\eta] \in A$ and $\eta \in A$, or $\neg[\eta] \in A$ and $\mu \in A$.*
– *A subset $A \subseteq cl(\varphi)$ of the closure is* unambiguous with respect to the until operator *if, for every $\eta\, \mathcal{U} \mu \in A$, either $\eta \in A$ and $\neg[\mu] \in A$, or $\mu \in A$.*

The transition function δ of the measuring automaton computes the set of successor atoms analogously to the construction for deterministic PLTL; the difference is that the successor atoms are no longer unique and we maintain a set of atoms.

– For the initial state q_0, the successor $\delta(q_0, e)$ is the set of atoms that contain φ and are consistent with e, i.e., the atoms $t \in At_\varphi$ with φ and $t \cap AP = e$.
– For every other state $t = s \subseteq At_\varphi$, the successor $\delta(s,e)$ is the set of atoms t' with $t \xrightarrow{e} t'$ for some $t \in s$.

We observe that the only nondeterminism in the successor relation $\xrightarrow{e}$ is in the selection of the non-parametric subformulas; once the non-parametric formulas have been chosen, the parametric formulas to be included in an atom are determined.

Lemma 9. *Let t, t' be two atoms in a state of the measuring automaton reached after reading some trace $\sigma = e_0 e_1 e_2 \ldots e_n$, such that there exists a sequence $t_1 t_2 \ldots t_{n+1}$ of atoms with $t_i \in s_i$ for $i = 1 \ldots n+1$ and $t_i \xrightarrow{e_i} t_{i+1}$ for $i = 1 \ldots n$ such that $t_{n+1} = t$, and a sequence $t'_1 t'_2 \ldots t'_{n+1}$ of atoms with $t'_i \in s_i$ for $i = 1 \ldots n+1$ and $t'_i \xrightarrow{e_i} t'_{i+1}$ for $i = 1 \ldots n$ such that $t'_{n+1} = t'$. If t_i and t'_i agree on the non-parametric formulas for all $i = 1 \ldots n+1$, then $t = t'$.*

The set X of variables contains now the variables $n_{x,t}$ and $m_{x,t}$ for each parameter $x \in V$ and each atom $t \in At_\varphi$, where the intended meaning is the same as in the construction for deterministic PLTL: the variable $n_{x,t}$ is the counter, the variable $m_{x,t}$ maintains the greatest counter value reached so far if x is an upward-closed parameter, and the smallest counter value reached so far if x is a downward-closed parameter.

As before, θ initializes n_x with 0, and m_x with ∞ for $\Box_{\leq x}$ and with 0 for $\Diamond_{\leq x}$. The key observation that allows us to define the update function γ is that every atom has a unique predecessor: while, for a pair of successor states s, s', each atom $t \in s$ may have multiple atoms $t' \in s'$ such that $t \xrightarrow{e} t'$, we have that, reversely, each atom $t' \in s'$ has exactly one atom in s with $t \xrightarrow{e} t'$; we denote this unique atom by $pre(s, s', t')$.

Lemma 10. *Let t_1, t_2 be two atoms in a state of the measuring automaton reached after reading some trace σ, and let t' be an atom in the state reached after reading the additional event e, such that $t_1 \xrightarrow{e} t'$ and $t_2 \xrightarrow{e} t'$. Then $t_1 = t_2$.*

The update function γ computes the new values of $n_{x,t}$ and $m_{x,t}$ based on the values of $n_{x,pre(s,s',t')}$ and $m_{x,pre(s,s',t')}$, i.e.,

```
for $\Diamond_{\leq x}\eta \in t'$: if $\eta \in t'$ then $m_{x,t'} := \max(m_{x,pre(s,s',t')}, n_{x,pre(s,s',t')}); n_{x,t'} := 0$
                 else $m_{x,t'} := m_{x,pre(s,s',t')}$; $n_{x,t'} := n_{x,pre(s,s',t')} + 1$;

for $\Box_{\leq x}\eta \in t'$: if $\eta \in t'$ then if $generated(\Box_{\leq x}\eta, pre(s,s',t'), t')$
                              then $m_{x,t'} := m_{x,pre(s,s',t')}; n_{x,t'} := 0$
                              else $m_{x,t'} := m_{x,pre(s,s',t')}; n_{x,t'} := n_{x,pre(s,s',t')} + 1$;
                 else $m_{x,t'} := \min(m_{x,pre(s,s',t')}, n_{x,pre(s,s',t')}); n_{x,t'} := 0$.
```

If x does not occur in the atom, then $m_{x,t'} := m_{x,pre(s,s',t')}$ and $n_{x,t'} := n_{x,pre(s,s',t')}$. The final states of the measuring automaton are the sets that contain an atom without unfulfilled obligations, i.e., $f \in F$ iff there is a $t \in f$ such that for all $\eta\, \mathcal{U} \mu \in t$ also $\mu \in t$, for all $\eta\, \mathcal{R}_\mu \in t$ also $\eta \in t$, and there is no formula $\bigcirc\psi$ in t. As the following lemma clarifies, every reachable final state $f \in F$ contains in fact exactly one such atom.

Lemma 11. *Every final state $f \in F$ reached by the measuring automaton on some trace contains exactly one atom $t \in f$, without unfulfilled obligations, i.e., where for all $\eta\, \mathcal{U} \mu \in t$ also $\mu \in t$, for all $\eta\, \mathcal{R}_\mu \in t$ also $\eta \in t$, and there is no formula $\bigcirc\psi$ in t.*

The output is based on this unique atom $t \in f$: ω reports the parameter assignment $x \mapsto m_{x,t}$ for all parameters, with the exception of remaining formulas of type $\Box_{\leq x}\mu$, where it reports $\min(m_{x,t}, n_{x,t})$ to take care of a new possible minimum on the last event on the trace.

Theorem 5. *For every PLTL formula φ there exists a measuring automaton $\mathcal{A}_\varphi = (\Sigma, \Omega, Q, q_0, X, \theta, \delta, \gamma, F, \omega)$ with an exponential number of states Q in $|\varphi|$ and a linear number of variables X in $|\varphi|$ such that for every sequence $\sigma \in (2^{AP})^*$, σ is accepted by $\mathcal{A}_\varphi$ with result r iff r is the measure of φ on σ under the unambiguous semantics.*

Soundness. The proof of the correctness of the construction of $\mathcal{A}_\varphi$ in Theorem 5 follows the structure of the proof of Theorem 3 for the corresponding construction for deterministic PLTL. The key difference is in the proof of Lemma 7, where we claim that for every trace σ and formula φ, if $(\sigma, 0, r) \models \varphi$, then there exists an atom sequence that satisfies the successor relation. For deterministic PLTL, this sequence can be constructed in a simple induction, progressing from the first position forwards, because the semantics is deterministic, i.e., the subformulas of the successor atom are uniquely determined by the present atom and the next event. For unambiguous PLTL, the parametric subformulas are chosen based on the truth value of the non-parametric subformulas. We therefore construct the sequence of atoms in two steps. In the first step, we compute, progressing backwards from the final position, precisely the set of non-parametric formulas that are satisfied in each position. In the second step, we add, progressing forwards from the initial position, the parametric subformulas according to the (now deterministic) semantics of unambiguous PLTL.

From Theorem 5 and Lemma 1 it follows that the space required by the online monitor is exponential in the size of the specification and logarithmic in the length of the trace.

Theorem 6. *Under the unambiguous semantics, a PLTL formula φ can be measured in exponential space in the size of φ and logarithmic space in the length of the trace.*

6 Experiments

We have implemented the offline measuring algorithm for PLTL from Section 3 and the online algorithm for unambiguous PLTL from Section 5. (Since unambiguous PLTL is a generalization of deterministic PLTL, the online algorithm handles the deterministic case as well.) The offline algorithm traverses the trace several times in backwards direction from the last event to the first event; the online algorithm traverses the trace just once in forward direction. Table 1 shows data from two experiments carried out with our Java implementation on a Intel Core i7 processor with 2.6 GHz and 8 GB main memory. The traces were generated based on simulation runs from two applications, a bus arbiter and a memory controller, and stored on a solid-state disk drive. We tested traces of varying length, between 10 000 and 10 000 000 events, and report the running time of both algorithms in milliseconds.

Bus arbiter. In this benchmark, we measure traces generated from an implementation of a synchronous bus arbiter for three clients. The parameters measure the duration between the occurrence of a request until a grant is given to the client.

Memory controller. Our second benchmark is a memory controller. The memory controller provides a bus interface to a memory module. We measure the retention period of a memory cell over a trace.

The online algorithm is significantly faster than the offline algorithm. In part, this can be explained by the fact that the offline algorithm has to perform several passes over the data, which is read anew from the disk every time. Additionally, the algorithm needs to evaluate all subformulas for every event, because a subformula might be satisfied from an earlier position onwards. The online algorithm only needs to evaluate the reachable sets of atoms and thus typically performs less work per event.

Table 1. Running times for the offline and online measuring algorithms

	bus arbiter		memory controller	
trace length	offline	online	offline	online
10k events	2027 ms	74 ms	444 ms	84 ms
100k events	6679 ms	263 ms	752 ms	246 ms
1M events	63711 ms	1484 ms	6275 ms	727 ms
10M events	180281 ms	13642 ms	65209 ms	15606 ms

7 Conclusions

We have presented a logical and algorithmic framework for *runtime measuring*: a way to provide quantitative results for a trace that needs to conform with an LTL specification, based on duration parameters attached to the LTL subformulas. This is a declarative approach, where minimal/maximal results need to be reported, as opposed to an operational approach, where counters are spawned and updated explicitly while the LTL formula is being verified. The complexity of runtime measuring depends on the disjunctive nature of a formula. Due to explicit disjunctions and temporal operators with implicit disjunctive semantics, such as *until*, there can be multiple answers. Disambiguating the results using priority among the measured variables resulted in an algorithm that requires quasilinear space in the length of the trace.

We showed that an efficient online algorithm is possible if one determinizes the measuring semantics of PLTL. The completely deterministic semantics of deterministic PLTL leads to a logarithmic space requirement in the length of the trace and a linear space requirement in the size of the specification; the combination of standard LTL semantics for non-parametric formulas with deterministic measuring in unambiguous PLTL increases the space requirement to exponential in the size of the specification, but still maintains the logarithmic dependency on the length of the trace.

Our interpretation is based on finite sequences, and thus differs a bit from the usual LTL interpretation. Indeed, in runtime verification, only a finite part of the sequence is always revealed. This is consistent with the formation of runtime verification algorithms for LTL [8,6]. A promising direction for future work is to consider an interpretation over infinite sequences. Then, we obtain three possible result values instead of the two Boolean values: *true*, *false* and *maybe* [10]. In the last position of the trace, we need to check whether the formulas in any of the closures are still satisfiable. Such a satisfiability algorithm would be similar to the one presented in [2].

Acknowledgments. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`). The first author was supported by an IMPRS-CS PhD Scholarship. The third author was supported by ISF grant 126/12 "Efficient Synthesis Method of Control for Concurrent Systems".

References

1. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime Monitoring of Synchronous Systems. In: TIME 2005, pp. 166–174 (2005)
2. Alur, R., Etessami, K., La Torre, S., Peled, D.: Parametric temporal logic for "model measuring". ACM Trans. Comput. Log. 2(3), 388–407 (2001)
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Software Engineering Methodologies 20(4), 14 (2011)
4. Etessami, K.: A note on a question of Peled and Wilke regarding stutter-invariant LTL. Information Processing Letters 75, 261–263 (2000)
5. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV 1995, pp. 3–18 (1995)
6. Havelund, K., Roşu, G.: Synthesizing Monitors for Safety Properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
7. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: Collecting statistics over runtime executions. Proceedings of Runtime Verification (RV 2002), Electronic Notes in Theoretical Computer Science 70(4), 36–54 (2002)
8. Finkbeiner, B., Sipma, H.: Checking Finite Traces Using Alternating Automata. Formal Methods in System Design 24(2), 101–127 (2004)
9. Kupferman, O., Piterman, N., Vardi, M.Y.: From Liveness to Promptness. Formal Methods in System Design 34(2), 83–103 (2009)
10. Kupferman, O., Vardi, M.Y.: Model Checking of Safety Properties. Formal Methods in System Design 19(3), 291–314 (2001)
11. Maidl, M.: The Common Fragment of CTL and LTL. In: FOCS 2000, Rodendo Beach, CA, USA, pp. 643–652 (2000)
12. Manna, Z., Pnueli, A.: Completing the Temporal Picture. Theoretical Computer Science 83, 91–130 (1991)
13. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
14. Rosu, G., Chen, F.: Semantics and Algorithms for Parametric Monitoring. Logical Methods in Computer Science 8, 1 (2012)
15. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, Cambridge, MA, pp. 332–344 (1986)

Precisely Deciding Control State Reachability in Concurrent Traces with Limited Observability

Chao Wang and Kevin Hoang

Department of ECE, Virginia Tech, Blacksburg, VA 24061, USA

Abstract. We propose a new algorithm for precisely deciding a *control state reachability (CSR)* problem in runtime verification of concurrent programs, where the trace provides only limited observability of the execution. Under the assumption of limited observability, we know only the type of each event (read, write, lock, unlock, etc.) and the associated shared object, but not the concrete values of these objects or the control/data dependency among these events. Our method is the first *sound and complete* method for deciding such CSR in traces that involve more than two threads, while handling both standard synchronization primitives and *ad hoc* synchronizations implemented via shared memory accesses. It relies on a new *polygraph* based analysis, which is provably more accurate than existing methods based on lockset analysis, acquisition history, universal causality graph, and a recently proposed method based the *causally-precedes* relation. We have implemented the method in an offline data-race detection tool and demonstrated its effectiveness on multithreaded C/C++ applications.

1 Introduction

The idea of using an offline analysis of the trace log to predict subtle bugs of a concurrent system has been the focus of intense research in recent years. The core problem in this analysis is to decide an instance of the *control state reachability (CSR)* problem: given a valid execution trace ρ, decide whether there exists an *alternative interleaving* ρ' of the events of the trace that can lead to a bad system state, e.g. one that manifests a data-race, a deadlock, or an atomicity violation. Although there exists a large body of work on trace based analysis for predicting concurrency bugs, e.g. using either underapproximation [19,20,7,24] or overapproximation [18,26,31,4], none of the existing methods can precisely decide CSR for input traces with *limited observability*.

Under the assumption of limited observability, the input trace records only the *global* operations but not *thread-local* operations. Even for the global operations, such as thread synchronizations and shared memory accesses, we only know the event types and the associated shared objects, but not the concrete values of these objects or the control/data dependency among the events. For instance, executing the code `tmp:=X; Y:=tmp+10;` would produce events `READ(X)` and `WRITE(Y)` in the trace log. However, we would not know the concrete values of X and Y, or whether `WRITE(Y)` is data-dependent on `READ(X)`. As another example, executing the code `if(X==10) Y:=0` would produce the same `READ(X)` and `WRITE(Y)` events in the trace log.

Precisely deciding CSR under limited observability is challenging for two reasons. First, we need to characterize the set of bugs that can be predicted, with certainty, by analyzing only the input trace under limited observability. Second, we need to design a new algorithm that can produce the exact set of *predictable* bugs. In other words, the algorithm should report a bug in the trace if and only if the bug is guaranteed to show up in some actual program execution. However, to the best of our knowledge, there does not exist any method that can precisely decide this CSR problem. For example, classic methods based on lockset analysis [18] may produce false alarms due to overapproximation, whereas classic methods based on the happens-before causality relation [13] may miss real bugs due to underapproximation.

For two threads synchronizing via nested locks only, Kahlon *et al.* [9] proposed the first sound and complete algorithm for deciding CSR based on a lock acquisition history (LAH) analysis. They subsequently proposed a lock causality graph (LCG) analysis [8], which generalizes LAH to handle also non-nested, but finite-length, lock chains. The theoretical significance of LAH and LCG is that they prove the decidability of CSR under certain synchronization patterns even for concurrently running recursive procedures. However, neither LAH nor LCG considers synchronization primitives other than locks. Kahlon and Wang [10] proposed a universal causality graph (UCG) analysis, which generalizes LCG to handle non-lock synchronization primitives as well. However, UCG is sound and complete for deciding CSR in traces that involve only two threads. For more than two threads, UCG may produce false alarms.

In this paper, we propose the first sound and complete algorithm for deciding CSR for input traces that involve more than two threads, while precisely handling both standard synchronizations and *ad hoc* synchronizations via shared memory accesses. We introduce a new *polygraph* based analysis framework, which is provably more accurate than the existing UCG analysis. The polygraph abstraction allows us to strengthen the CSR decision procedure to make it sound and complete for traces that involve an arbitrary, but fixed, number of threads. Note that, being part of a testing procedure, our method do not guarantee to detect all bugs in the program. Instead, our *sound and complete* argument is restricted to the set of *predictable* bugs for the given input execution trace.

We also introduce a new *predictive model* to more accurately model not only standard synchronization primitives such as locks and wait/notify (as in UCG), but also *ad hoc* synchronizations implemented using shared memory accesses. This can significantly increase the precision of bug detection. When being applied to data-race detection, for instance, our method will be provably more accurate than both UCG and a more recent *causally precedes* (CP [24]) analysis. Similar to ours, the CP method does not report false alarms. However, it accomplishes this by aggressively dropping valid interleavings, in a way that may lead to missed bugs. Our method does not have this problem.

We have implemented the new method in an offline data-race detection tool based on the LLVM platform. Our experiments conducted on a set of multithreaded C/C++ applications show that the new method is indeed more accurate that the existing ones.

To sum up, we have made the following contributions:

– We propose a new *polygraph* based analysis for precisely deciding CSR in execution traces with only limited visibility. The method is sound and complete for an arbitrary, but fixed, number of concurrent threads.

- We implement the new method inside an offline data-race detection tool and demonstrate that, in addition to being provably more accurate, it actually reports more real bugs than CP and fewer false alarms than UCG.

2 Preliminaries

The control state reachability (CSR) problem arises from trace based analysis of multithreaded programs for detecting subtle concurrency bugs. Given a concrete execution trace ρ of the program, this analysis typically consists of three steps:

1. Identify a set of *potential* bugs by scanning the trace ρ for known error patterns.
2. For each potential bug, create an error condition EC and check for *feasibility*, i.e. EC can be satisfied by some valid interleaving ρ' of the concurrent events in ρ.
3. Compute a thread schedule for each bug found in Step 2 for deterministic replay.

Our main contribution lies in step 2, where our new algorithm can precisely decide the feasibility of EC. This is equivalent to deciding the CSR problem, where we are concerned with the simultaneous reachability of some locations in the concurrent threads communicating through standard and *ad hoc* synchronization operations.

Let the input trace be a sequence $\rho = e_1, \ldots, e_n$, where each event e_i $(1 \leq i \leq n)$ models an operation in thread T_i of one of the following types:

- $fork(thrd)$ for the creation of child thread $thrd$; and
- $join(thrd)$ for the join back of child thread $thrd$.
- $acq(lk)$ for acquiring lock lk;
- $rel(lk)$ for releasing lock lk;
- $signal(cv)$ for sending signal via condition variable cv;
- $wait(cv)$ for receiving signal via condition variable cv;
- $R(sh)$ for reading from shared variable sh;
- $W(sh)$ for writing to shared variable sh.

For example, the wait operation in POSIX threads is of the form $wait(cv, lk)$, where cv is a condition variable and lk is a lock. Based on the POSIX standard, this operation consists of three substeps. First, the thread releases lk to allow other threads to acquire lk and execute $signal(cv)$. Next, the thread enters the sleep mode. Finally, after the signal operation is executed by another thread, the thread wakes up and acquires lk again. There, the entire wait operation is equivalent to $rel(lk); wait(cv); acq(lk)$.

We assume that the input trace ρ is feasible because it represents a real execution. If ρ itself exposes a bug, we are done. Otherwise, we check whether there exists a permutation ρ' that exposes a bug. Permutation ρ' is feasible (or real) if ρ' can appear in some actual execution of the program. Furthermore, we assume that it is not possible to run the original program again to test the validity of ρ'. Instead, we define a statically checkable condition over ρ, under which permutation ρ' is guaranteed to be feasible.

First, we define the condition for threads synchronizing via standard synchronization primitives. Let $\rho = e_1, \ldots, e_n$ be the input trace and $\rho' = e'_1, \ldots, e'_n$ be a permutation, where for all $1 \leq i, j \leq n$, each event e'_i maps to a unique e_j and vice versa. Let $e_i \rightarrow e_j$ denote that e_i appears before e_j when the two events are in different threads. Let $e_i <_{PO} e_j$ denote that e_i appears before e_j in the same thread. The conditions for ρ' is to be feasible are as follows (c.f. [10]):

1. **program order**: events within each thread must follow their program order. That is, $e_i <_{PO} e_j$ if e_i appears before e_j in ρ and both events are from the same thread.
2. **fork/join order**: events in a child thread t must appear after the $fork(t)$ event, but before the $join(t)$ event, of the parent thread.
3. **signal/wait order**: events in each matching $signal(cv)$ and $wait(cv)$ pair must appear in the same order as they appear in the input trace ρ.
4. **acq/rel order**: events from two matching lock/unlock pairs, e.g. (acq_1, rel_1) and (acq_2, rel_2) over the same lock, must be mutually exclusive. Since critical sections should not interleave, either $rel_1 \rightarrow acq_2$ or $rel_2 \rightarrow acq_1$.

For threads that do not synchronize via shared memory accesses, these are both sufficient and necessary conditions for ρ' to be feasible. However, in the general case, they are only necessary conditions. That is, if any condition is violated, ρ' is guaranteed to be infeasible. But even if all of them are satisfied, ρ' may still be infeasible. It is worth pointing out that, since the input trace ρ is feasible, every lock acquired by a thread in ρ must have been released by the same thread. To overcome the problem, we add the following condition:

5. **write/read order**: events from two matching write/read pairs, e.g. (W_1, R_1) and (W_2, R_2), where $R_1(x)$ reads the value set by $W_1(x)$ and $R_2(x)$ reads the value set by $W_2(x)$, must not interfere. They should satisfy $W_1 \rightarrow R_1$, $W_2 \rightarrow R_2$, and in addition, either $R_1 \rightarrow W_2$ or $R_2 \rightarrow W_1$.

This condition may not hold in all execution traces, but instead, is imposed by the given input trace ρ. The condition ensures that, as long as ρ is feasible, ρ' is also feasible, even if we do not have any information about the program that generates the input trace ρ. If there exist a write event in ρ that does not have any matching read event, we add a dummy read event immediately after the write event.

We assume that ρ provides limited observability of the program. For example, we do not know whether the read event $R(x)$ comes from `a:=x+5` or `if(x>10)` or `if(x<0)`. Similarly, we do not know whether the write event $W(x)$ comes from `x:=10` or `x:=0`. Given the sequence $R(x) \ldots W(sh)$, we do not know whether $W(sh)$, which may come from `a:=sh`, is control-dependent on $R(x)$, which may come from `if(x>0)`. Assume that `a` and `b` are thread-local, traces from the following programs are indistinguishable:

– **program 1:** `if (x==0) { b:=0; } a:=sh;` or
– **program 2:** `if (x!=0) { a:=sh; }` or
– **program 3:** `{ b:=x; a:=sh; }`

Nevertheless, we show that, by requiring each $R_1(x)$ in ρ' to read from the same $W_1(x)$ as in ρ, we can ensure that $R(sh)$ will be executed in ρ', regardless of the expression in the if-condition, and whether the if-condition is guarding $R(sh)$.

We want to stress that the core analysis procedure proposed in this paper is not tied up to whether the write/read order (Condition 5) is used or not. To make our subsequent presentation clear, we define the following two types of predictive models:

– The **CSR model** includes Conditions 1, 2, 3, and 4, but not Condition 5.
– The **CDSR model** includes Conditions 1, 2, 3, 4, and 5.

The CSR model considers only standard synchronizations, whereas the CDSR model also considers *ad hoc* synchronizations implemented by using shared memory accesses. In the sequel, we shall present our new polygraph based analysis method for the CSR model first, and then extend it the CDSR model.

3 Polygraph Based Causality Analysis

Deciding the feasibility of an error condition EC is challenging mainly due to *interleaving explosion* – the number of possible interleavings is often exponentially large. Therefore, naively enumerating the feasible interleavings and checking them against EC is not practical. Instead, we rely on checking a new *polygraph* where deciding the feasibility of EC is equivalent to deciding the absence of cycles in the graph.

3.1 From Input Trace ρ to Polygraph G_ρ

A *polygraph* is a generalization of a directed graph that we use to capture all feasible interleavings of the events of an input trace. The term was coined by Papadimitriou [16] while studying view serializability: nodes in his polygraph are requests and responses of database transactions, whereas in our case, they are events of a multithreaded program. Let the input trace be ρ. The ρ-induced polygraph, denoted $G_\rho = (V, E, E_{poly})$, consists of a set V of nodes, a set E of edges, and a set E_{poly} of polyedges:

- Each node in V models an event in ρ.
- Each edge in E, denoted $a \dashrightarrow b$, means a must appear before b. Initially, these edges come from the program order, fork/join, and signal/wait as defined in Section 2.
- Each polyedge in E_{poly}, denoted $(a \dashrightarrow b, c \dashrightarrow d)$, represents an *either-or* choice, meaning that either a appears before b, or c appears before d.

For the CSR model (Section 2), the polyedges come from the **acq-rel** event pairs:

- For any two *acq-rel* event pairs over lock lk, say (acq_1, rel_1) and (acq_2, rel_2), their mutual exclusion demands that either rel_1 appears before acq_2, or rel_2 appears before acq_1. In G_ρ, this is modeled by polyedge $(rel_1 \dashrightarrow acq_2, rel_2 \dashrightarrow acq_1)$.

This defines the set of interleavings in the CSR model, for which we set out to design a sound and complete algorithm for checking the feasibility of EC.

Later, in Section 5, we will add another type of polyedges for modeling the non-interference properties of the write-read pairs (the CDSR model in Section 2). If the threads synchronize solely via standard synchronization primitives – without using shared variable accesses – the CSR model would be precisely for predicting all the real bugs (w.r.t. the input trace). Otherwise, we need to consider the CDSR model. Regardless, our core analysis procedure works for both models.

3.2 From Error Condition EC to Polygraph $G_\rho(EC)$

Given an error condition EC, e.g. representing a potential data-race, we construct a new polygraph $G_\rho(EC)$, which is a specialization of G_ρ for checking EC. We model

EC by adding a set of new edges to G_ρ, and then perform a slicing of G_ρ, to remove all polyedges that are irrelevant to satisfying EC. Below are examples for modeling some typical concurrency bugs:

- *Data-race*: Let (a, b) be the events that have a potential data-race. These potential data-races may be computed using standard lockset analysis [18]. That is, we compute the set of locks held by a thread at each of its program locations. Then, for any two events a and b that access the same memory from different threads without holding a common lock, there is a potential data-race. Let a' and b' be the immediate preceding events of a and b in the two threads, respectively. The error condition EC consists of edges $a' \rightarrow b$ and $b' \rightarrow a$.
- *Atomicity violation*: Let a and b be two events that are intended to execute atomically in one thread, and c be an interfering event in another thread. Here, c interferes with a (and b) if and only if they access the same memory location with at least one write event. In such case, the event order $a < c < b$ indicates a violation. The error condition EC consists of edges $a \rightarrow c$ and $c \rightarrow b$.
- *Order violation*: Let the event sequence $a_1, \ldots, a_k$ be an unintended execution order. The error condition EC consists of edges $a_1 \rightarrow a_2$, $a_2 \rightarrow a_3$ ..., and $a_{k-1} \rightarrow a_k$, since the violation is exposed when all these edges are satisfied.

Slicing $G_\rho(EC)$ with respect to EC broadens the coverage and allows more real bugs to be detected. Recall that our goal is to find a valid interleaving that can lead to the satisfaction of all edges in EC. However, the valid interleaving does not have to be a permutation of the whole input trace ρ. Instead, a subsequence or prefix, from the initial state to EC, would suffice. Therefore, we remove from $G_\rho(EC)$ any happens-before obligation (edges and polyedges) that are not needed for satisfying EC.

3.3 Resolving the Polyedges to Detect Cycles

Since each edge in $G_\rho(EC)$ represents a *precedence* relation that must be satisfied, a cycle in this graph means that no valid interleaving exists. Therefore, we decide the feasibility of EC by detecting cycles in $G_\rho(EC)$. Sometimes, $G_\rho(EC)$ has no cycle initially, but existing edges may force the *either-or* decisions of some polyedges, which in turn lead to cycles. This *polyedge resolution* process is often triggered by the addition of the EC edges. It is iterative because resolving one polyedge may introduce a new regular edge that triggers the resolution of another polyedge.

For the CSR model (Section 2), we use the following polyedge resolution rule:

Rule 1: For any polyedge $(rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1)$, if there already exists a regular edge $s \rightarrow t$ such that $acq_1 <_{PO} s$ and $t <_{PO} rel_2$, we remove the polyedge and add regular edge $rel_1 \rightarrow acq_2$ (see the three cases in Fig. 1). The reason is that, the other choice $rel_2 \rightarrow acq_1$ would have formed a cycle with $s \rightarrow t$.

Therefore, our procedure starts by resolving all polyedges whose choices are forced by some existing edges. It repeats the process until (1) a cycle is detected, meaning that no feasible interleaving exists in $G_\rho(EC)$; (2) all polyedges are resolved, meaning that a feasible interleaving is found; or (3) there are still some unresolved polyedges. In the third case, the problem remains undecided.

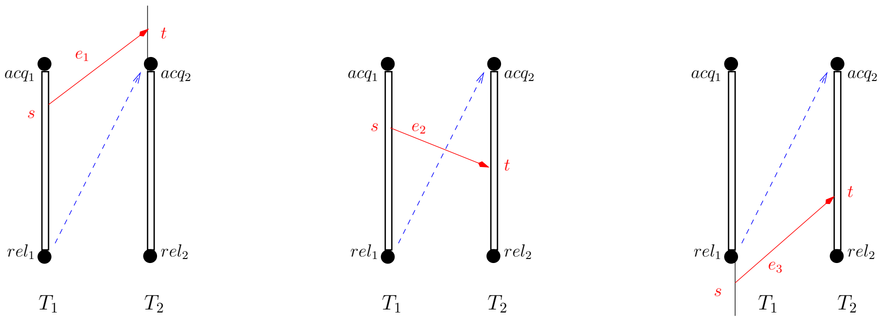


Fig. 1. Polyedge resolution due to existing edge $s \to t$, which adds $rel_1 \to acq_2$

By now, our new polygraph based procedure matches the precision of the UCG [10], although the underlying decision mechanisms are drastically different. UCG does not rely on polygraph, but instead on a set of custom made inference rules. Our use of polygraph allows the new analysis framework to be easily extended to handle not only standard synchronization primitives but also *ad hoc* synchronizations such as shared memory read/write accesses.

If the goal is to design an over-approximated analysis, the undecided EC in Case 3 may be reported as a potential bug. If the goal is to design an under-approximated analysis, the undecided EC in Case 3 may be dropped. Both would lead to a loss of precision. In the next section, we shall propose a new method for resolving the third case, thereby avoiding the precision loss.

4 Generalizing the Algorithm to *k* Threads

We start by proving that the polyedge resolution algorithm in Section 3.3 (Rule 1) is sound and complete for two threads communicating via standard synchronizations. Then, we show why it does not work for traces with more than two threads. Finally, we extend Rule 1 to make it sound and complete for traces with more than two threads.

Theorem 1. *If our algorithm defined in Section 3.3 generates a cycle in $G_\rho(EC)$, there does not exist any valid interleaving in the CSR predictive model that satisfies EC.*

The proof is straightforward because all edges in $G_\rho(EC)$ represent *precedence* relations that must hold at all time. Thus, a cycle meaning that EC is not satisfiable. □

Theorem 2. *If our algorithm defined in Section 3.3 does not generate a cycle in $G_\rho(EC)$, for 2 threads, a valid interleaving always exists in the CSR predictive model.*

The proof consists of two cases. First, if all polyedges are resolved at the end of the iterative process and there is no cycle, since nondeterminism is removed completely, EC is satisfiable. Second, if some polyedges still remain un-resolved at the end of the iterative process, we show that they can always be resolved as follows:

1. Pick a polyedge arbitrarily and replace it with one of the *either-or* edges.
2. Apply Rule 1 to resolve the affected polyedges.
3. Repeat the above steps until all polyedges are resolved.

We prove, by contradiction, that the above process would not create any cycle for two threads. Assume that resolving polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1\rangle$ into regular edge $rel_1 \rightarrow acq_2$ creates a cycle subsequently. With two threads, the cycle must involve edges $rel_1 \rightarrow acq_2 <_{PO} s \rightarrow t <_{PO} rel_1$, where $s \rightarrow t$ is an existing edge. However, based on Rule 1, since edge $s \rightarrow t$ already exists, it should have resolved the polyedge already. Therefore, our assumption is not correct. The theorem is proved. □

A byproduct is that, for two threads, the *schedule reconstruction* procedure as described above can always generate a valid thread interleaving in polynomial time.

4.1 From 2 Threads to 3 Threads

The proof in Theorem 2 does not work for trees with 3 threads, as shown in Fig. 2. Here, we have four regular edges ($s \rightarrow p$, $s' \rightarrow p$, $q \rightarrow t'$, and $q \rightarrow t$) and one polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1\rangle$. While there is no feasible interleaving, there is no cycle either. The reason why there is no feasible interleaving is due to the transitive precedence constraints $s \rightsquigarrow t$ and $s' \rightsquigarrow t'$. However, notice that none of the regular edges alone can resolve the polyedge based on Rule 1. Furthermore, both choices of the polyedge would create a cycle.

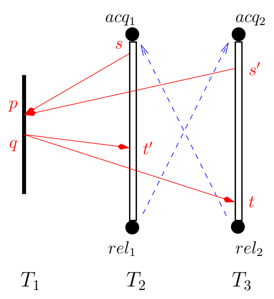


Fig. 2. Example (3 threads): There is no valid interleaving and the polygraphs have no cycle

A straightforward way to strengthen the algorithm is to include transitive edges such as $s \rightsquigarrow t$ in Rule 1. For example, in Fig. 2, one transitive edge is $s \rightarrow p <_{PO} q \rightarrow t$ and another is $s' \rightarrow p <_{PO} q \rightarrow t'$. With the modified Rule 1, these two transitive edges would lead to a cycle. Unfortunately, although this fix works for traces with 3 threads, it does not work for traces with 4 threads, as shown by Fig. 3.

Fig. 3 has two polyedges that cannot be resolved by the existing edges and their transitive edges. Therefore, the polygraph is cycle-free. However, there does not exist a feasible interleaving either. Consider all four cases for resolving the two polyedges – all would lead to contradictions (cycles):

- If we select $rel'_1 \rightarrow acq'_2$ from the second polyedge, transitive edge $s \rightarrow q_1 <_{PO} rel'_1 \rightarrow acq'_2 <_{PO} p_2 \rightarrow t$ would induce $rel_1 \rightarrow acq_2$; bug transitive edge $s' \rightarrow q_1 <_{PO} rel'_1 \rightarrow acq'_2 <_{PO} p_2 \rightarrow t'$ would induce $rel_2 \rightarrow acq_1$.
- If we select $rel'_2 \rightarrow acq'_1$ from the second polyedge, transitive edge $s \rightarrow q_2 <_{PO} rel'_2 \rightarrow acq'_1 <_{PO} p_1 \rightarrow t$ would induce $rel_1 \rightarrow acq_2$; but transitive edge $s' \rightarrow q_2 <_{PO} rel'_2 \rightarrow acq'_1 <_{PO} p_1 \rightarrow t'$ would induce $rel_2 \rightarrow acq_1$.

Therefore, we need to strengthen the algorithm further for traces with 4 or more threads.

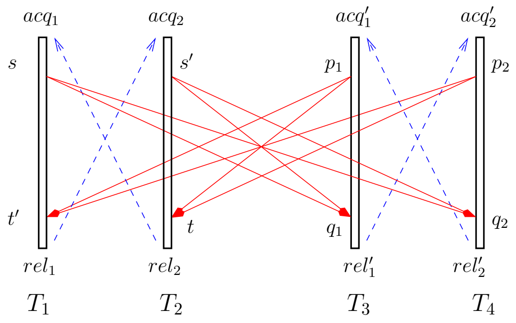


Fig. 3. Example (4 threads): There is no valid interleaving and the polygraphs have no cycle

4.2 Heuristics for Resolving the Remaining Polyedges

Before continuing our effort on strengthening Rule 1, let us pause for a moment to consider the current polygraph $G_\rho(EC)$, which (1) has no cycle, and also (2) has some unresolved polyedges. Therefore, either it has a feasible interleaving, or it does not. Recall that Rule 1 is geared toward proving infeasibility (by finding a cycle). What if EC actually is feasible? In this case, the best strategy is not to find a cycle (since it does not exist) but to find a valid interleaving. Toward this end, we must resolve the remaining polyedges in a consistent fashion, for example, by employing our *schedule reconstruction* algorithm proposed as part of the proof for Theorem 2.

Recall that in the proof for Theorem 2, for each unresolved polyedge, we arbitrarily pick one of the either-or choices and then apply Rule 1 to propagate its impact on the remaining polyedges. If we can resolve all polyedges without creating a cycle, we have proved the feasibility of the EC. However, since the *schedule reconstruction* algorithm resolves polyedges arbitrarily, it may not be the best strategy for finding a feasible interleaving if there exists one.

Fig. 4 shows that, if we make the wrong decision, by choosing $rel_1 \rightarrow acq_2$ first to resolve the polyedge on the left-hand side, it would lead to a cycle regardless of how we resolve the second polyedge. Notice that in this example, there actually exists a feasible interleaving: it is possible to resolve both polyedges, e.g. by picking $rel_2 \rightarrow acq_1$ and $rel'_2 \rightarrow acq'_1$, while avoiding the creation of any cycle.

Therefore, we use a *causally precedes (CP)* relation [24] as guidance to increase the success rate of the schedule reconstruction. That is, we impose a strict precedence order

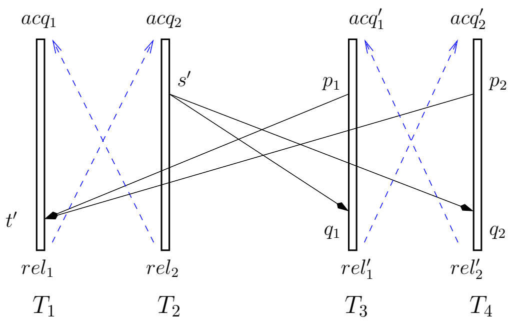


Fig. 4. There exists a valid interleaving, but arbitrarily selecting edges from unresolved polyedges, e.g. $rel_1 \rightarrow acq_2$, may lead to a cycle. Note that $s' \rightsquigarrow t'$ always holds.

between any two critical sections that share conflicting data accesses – two accesses of the same memory location and at least one of them is a write. Enforcing the CP relation takes a polynomial time w.r.t. the trace length, and the main advantage is that, if the graph remains acyclic, EC is guaranteed to be feasible (c.f. [24]). More specifically, our use of the CP relation based heuristic is as follows:

– For any unresolved polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1 \rangle$, if $rel_1 <_{CP} acq_2$, where $<_{CP}$ means *causally-precedes* [24], we replace it with edge $rel_1 \rightarrow acq_2$;

If the above heuristic search finds a valid interleaving, we are done. Otherwise, we resolve it in the the next step. Therefore, our overall method is at least as accurate as the CP-only method [24] for detecting bugs. In Section 6, we will show that in practice, our method is often significantly better.

4.3 Generalizing the Resolution Rule for *k* Threads

When the CP heuristic based search in Section 4.2 encounters a cycle, it does not mean that EC is infeasible, since the cycle may be created by a wrong decision. Therefore, we should backtrack and try again. However, a naive backtracking algorithm can be expensive: for $|E_{poly}|$ polyedges, it may take $O(2^{|E_{poly}|})$ time.

Instead, we propose a bounded lookahead search whose complexity remains polynomial in $|E_{poly}|$. Let k be the number of threads. We propose to strengthen Rule 1 with an exhaustive lookahead of k polyedges, to explore both of the either-or choices of all k polyedges. The goal is to identify hidden implications such as the ones in Fig. 3. In particular, Rule 1 is modified as follows:

Rule 1 (strengthened): For any polyedge $\langle rel_1 \rightarrow acq_2, rel_2 \rightarrow acq_1 \rangle$, we also check if there exists a path from s to t such that (1) it involves $\leq k$ polyedges along the way and (2) all the 2^k ways of revolving this polyedges lead to $s < t$. If such path exists ($s \rightarrow t$ is only a special case), we remove the polyedge and add regular edge $rel_1 \rightarrow acq_2$, since it is implied.

This strengthened rule directly leads to the proof of the following theorem.

Theorem 3. *If our strengthened algorithm as defined in this section does not generate a cycle in $G_\rho(EC)$, a valid interleaving always exists in the CSR predictive model.*

It is polynomial in $|E_{poly}|$ for two reasons. First, in a graph with $|V|$ nodes, there are at most $O(|V|^2)$ edges to add, which bounds the number of iterations. Furthermore, adding one such edge requires a graph analysis which takes $O(|V|+|E|)$ time. Second, with k threads, we only need to check for cycles that involve at most k threads and therefore k polyedges, because larger cycles can be decomposed into these smaller cycles. Checking all possible combinations of k polyedges takes $O(2^k)$ time. With $|E_{poly}|$ polyedges, there are at most $O(|E_{poly}|^k)$ distinct cycles that need to be inspected in the k-step lookahead.

Therefore, the overall method takes $O(|V|^2\ (|V| + |E|)\ 2^k\ |E_{poly}|^k)$ time, which is polynomial in $|V|$ and E. Note that in an *offline* trace based analysis, the number of threads k is fixed, and often small, whereas the trace length $|V|$ can be arbitrarily large. Therefore, it is advantageous to have a worst-case complexity polynomial in $|V|$.

4.4 The Overall Flow

The overall algorithm is illustrated in Fig. 5. Given an input trace ρ, we first construct a polygraph G_ρ and compute a set of potential error conditions. For each error condition EC, we construct the specialized polygraph $G_\rho(EC)$ and resolve polyedges. If there exists a cycle in $G_\rho(EC)$, we conclude that the error condition cannot be satisfied. Otherwise, we search for a valid schedule using the CP heuristic. If a valid schedule is found, it is a real bug. Otherwise, we switch to the lookahead search.

When the number of events $|V|$ is large and the number of threads is more than 2, the lookahead search may become expensive. In such case, other practical *tricks* may be needed, together with our new algorithm, to control the execution time. For example, a popular technique in runtime verification of large applications is to restrict the analysis to a sliding window of say, 5000 events, as opposed to the entire trace.

Even in such case, our generalized algorithm is valuable for two reasons. First, it provides useful insight for us to understand the various sources of precision loss in the CSR analysis. Second, it provides a unified framework for us to progressively increase the precision of the CSR analysis in the practical implementation. For example, when we set the lookahead depth to 1, 2, 3, ..., our algorithm would become precise automatically for traces that involve an increasing number of threads.

5 Applying the New Algorithm to CDSR Model

For the second predictive model defined in Section 2, namely the CDSR model, our polygraph G_ρ contains not only the **acq-rel** polyedges (Section 3.2) but also another type of polyedges, called the **write-read** polyedges:

– For any two *write-read* event pairs, denoted (W_1, R_1) and (W_2, R_2), where $R_1(x)$ reads from $W_1(x)$, and $R_2(x)$ reads from $W_2(x)$ in ρ, we maintain their write-to-read correspondence by requiring that either R_1 appears before W_2, or R_2 appears before W_1. The polyedge is denoted $\langle R_1 \to W_2, R_2 \to W_1\rangle$.

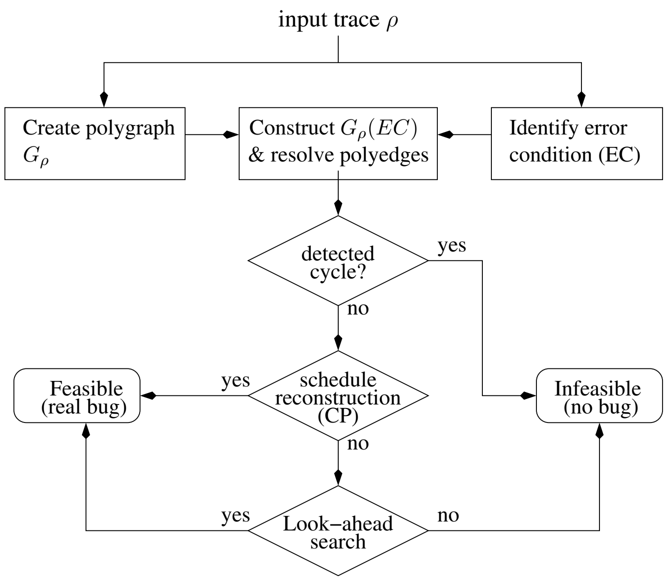

input trace ρ
Create polygraph G_ρ
Construct $G_\rho(EC)$ & resolve polyedges
Identify error condition (EC)
detected cycle?
yes
no
schedule reconstruction (CP)
yes
no
Feasible (real bug)
Infeasible (no bug)
Look–ahead search
yes
no

Fig. 5. Overall flow of our polygraph based prediction method

The new polyedges ensure that all the $R(x)$ events get the same $W(x)$ events as in the input trace ρ. Although this is not required by the program semantics, it is required by the CDSR model to ensure that ρ' is feasible whenever ρ is feasible.

Interestingly, there is an analogy between the *write-read* polyedge and the *acq-rel* polyedge. The non-interference property of write-read event pairs is similar in its form – although not in its meaning – to the mutual exclusion property of the critical sections defined by the acq-rel event pairs.

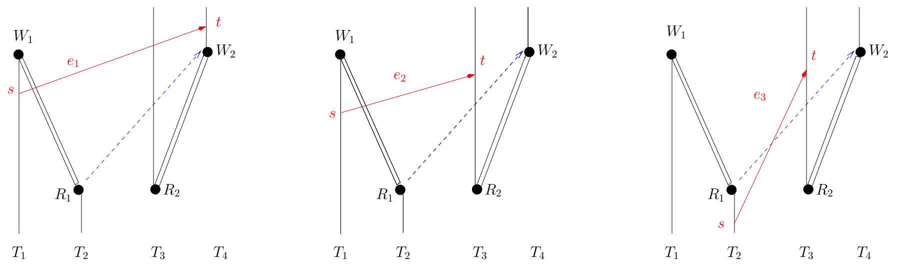


Fig. 6. Polyedge resolution due to existing edge $s \to t$, which adds edge $R_1 \to W_2$

Therefore, similar to Rule 1 in Section 3.3, we define a new polyedge resolution rule as follows:

Rule 2. For any polyedge $\langle R_1 \rightarrow W_2, R_2 \rightarrow W_1 \rangle$, we check if there exists an edge $s \rightarrow t$ in the graph that forces the resolution of the either-or choice.

- When each write-read event pair comes from the same thread, non-interference is similar to mutual exclusion – hence the new rule is the same as Rule 1 (see the three cases in Fig. 1), except for substituting acq with W and rel with R.
- When the events from a write-read pair come from two different threads, the rule is slightly different (see the three cases in Fig. 6). In all of the three cases, if there exists an edge $s \rightarrow t$ in the graph that forces the resolution of the either-or choice, we replace the polygraph with edge $R_1 \rightarrow W_2$. More specifically, we look for edge $s \rightarrow t$ such that $W_1 < s$ and $t < R_2$. There are two ways to satisfy $W_1 < s$. One is $W_1 <_{PO} s$ as illustrated by the source nodes of edges e_1 and e_2 in Fig. 6. The other is $R_1 <_{PO} s$, which together with $W_1 \rightarrow R_1$ leads to $W_1 < s$ as illustrated by the source node of edge e_3.

Similar to the original Rule 1, the above rule works only for two threads. To handle traces with more than two threads, we strengthen Rule 2 in the same way as we strengthen Rule 1 in Section 4.3. That is, in the **Strengthened Rule 2**, we check for not only an edge $s \rightarrow t$, but also a path from s to t such that (1) it involves $\leq k$ polyedges along the way and (2) all the 2^k ways of revolving these polyedges lead to $s < t$. If such a path exists, we must replace the polyedge $\langle R_1 \rightarrow W_2, R_2 \rightarrow W_1 \rangle$ with the regular edge $R_1 \rightarrow W_2$ since it is implied.

The proof of correctness is almost identical to the one for the CSR model and therefore is omitted. We give the two theorems as follows:

Theorem 4. *If our algorithm defined in this section generates a cycle in $G_\rho(EC)$, there does not exist any valid interleaving in the CDSR model that satisfies EC.*

Theorem 5. *If our algorithm does not generate a cycle in $G_\rho(EC)$, a valid interleaving always exists in the CDSR model.*

By now, our new polygraph based analysis is already provably more accurate than the UCG analysis [10] in the following sense. First, it works not only for two threads – as in UCG – but also for an arbitrary (but fixed) number of threads. Second, the CDSR model is more accurate than the one used in UCG. Recall that UCG has only inference rules that are equivalent to our **Rule 1**, whereas our new method also has **Rule 2**.

6 Running Examples

In this section, we demonstrate the application of our new decision procedure in an offline predictive analysis for detecting data-races. We use a popular programming idiom in POSIX threads to illustrate some application specific optimizations that we made during our implementation in contrast to both UCG and CP. The program in Fig. 7 (left) shows a typical scenario for using condition variable c, which ensures that assignment `x:=1` in thread T_1 always appears before `a:=x` in thread T_2. Here, lock l protects the concurrent accesses to the condition variable c since it is shared by both threads.

According to the POSIX standard, if T_2 enters the critical section ($l_7 \ldots l_{12}$) first, the execution of `wait(c,l)` would release lock l and block. At this time, thread T_1 has to execute `signal(c)` and then `unlock(l)` at l_6. After that, T_2 wakes up from `wait(c,l)`, re-acquires lock l and then continues. This thread interleaving is shown by Trace 1 in the middle, where the $wait(c, l)$ operation splits into three distinct events $rel(l)$; $wait(c)$; $acq(l)$.

Thread T_1	Thread T_2
$l_1 : x = 1;$	$l_7 : lock(l);$
$l_2 : y = 2;$	$l_8 : y = 3;$
$l_3 : lock(l);$	$l_9 : if(z == 0)$
$l_4 : signal(c);$	$l_{10} : \quad wait(c, l);$
$l_5 : z = 1;$	$l_{11} : a = x;$
$l_6 : unlock(l);$	$l_{12} : unlock(l);$

Two concurrent threads synchronize through the lock l, condition variable c, and shared variable z.

RHS are the two execution traces.

T_1	T_2
$l_1 : W(x)$	
$l_2 : W(y)$	$l_7 : acq(l)$
	$l_8 : W(y)$
	$l_9 : R(z)$
	$l'_{10} : rel(l)$
$l_3 : acq(l)$	
$l_4 : signal(c)$	
$l_5 : W(z)$	
$l_6 : rel(l)$	$l''_{10} : wait(c)$
	$l'''_{10} : acq(l)$
	$l_{11} : R(x)$
	$l_{12} : rel(l)$

trace 1

T_1	T_2
$l_1 : W(x)$	
$l_2 : W(y)$	
$l_3 : acq(l)$	
$l_4 : nop$	
$l_5 : W(z)$	
$l_6 : rel(l)$	
	$l_7 : acq(l)$
	$l_8 : W(y)$
	$l_9 : R(z)$
	$l_{11} : R(x)$
	$l_{12} : rel(l)$

trace 2

Fig. 7. UCG would report a bogus race on x on trace 2; CP would miss the real race on y

The use of variable z is crucial in ensuring that `wait(c,l)` is executed only when T_1 has not yet executed `signal(c)`. If T_1 enters the critical section ($l_3 \ldots l_6$) and executes `signal(c)` first, since T_2 is not waiting, the signal sent by T_1 would be lost. In this case, T_2 must skip `wait(c,l)` based on checking z's value. Otherwise, executing `wait(c,l)` would cause T_2 to hang. This second interleaving is shown in Trace 2.

Also note that the two input traces provide only limited observability. That is, we know l_8 is a write to y in trace 1 but not the value written (or the right-hand-side expression). We know that both l'_{10} and l_{11} happen after l_9, but do not know that l_{10} is guarded by l_9 but l_{11} is not. These are reasonable assumptions in many real-world applications, where getting more detailed program information is expensive or impossible, e.g. when the execution trace logs are generated during production runs on the client site.

There are two potential data-races on variables x and y, respectively. Here a data-race refers to the simultaneous accesses of the same memory location by two concurrent threads, where at least one access is a write. These potential data-races can be identified by a standard lockset analysis, because the thread locations $(1_1, l_{11})$ and (l_2, l_8) are not protected by the same lock.

When the input is Trace 1, both UCG and CP would correctly decide that the data-race on y is real and the data-race on x is bogus, due to the signal-to-wait edge $l_4 \to l''_{10}$. However, when the input is Trace 2, UCG would incorrectly classify (l_1, l_{11}) as being reachable (bogus) and CP would incorrectly classify (l_2, l_8) as unreachable (missed).

The imprecision of UCG for Trace 2 is due to the fact that UCG considers only standard synchronization operations while ignoring *ad hoc* synchronization events such as $W(z)$ and $R(z)$. Since l_1 and l_{11} are not protected by a common lock, and not ordered by signal/wait, UCG assumes that they can be executed simultaneously.

The imprecision of CP for Trace 2 is due to its dropping of valid interleavings too aggressively. That is, whenever two critical sections share conflicting data accesses – such as $W(z)$ and $R(z)$ in Trace 2 – it imposes a *causally-precedes* relation, effectively adding $l_6 <_{CP} l_7$. Although this removes the bogus data-race on x, it also removes the real data-race on y because it forbids any interleaving in which the order of the two critical section is swapped. Indeed, the constraint $l_5 <_{CP} l_6$ is too strong, because according to the semantics of the locks, l_6 can actually happen before l_5.

Our method, in contrast, can correctly classify both cases in Fig. 7. It improves over UCG by adding the new Rule 2. The addition of write/read consistency, in particular, is responsible for the correct classification of the data-race on x. It also improves over CP by applying the goal (EC) directed polyedge slicing (Section 3.2), goal directed polyedge resolution (Section 3.3), before resorting to the use of CP heuristic (Section 4.2). Therefore, it may be able to find a feasible solution without using the (often more restrictive) CP relation at all. Indeed, this is the reason why we can still detect the data-race on y whereas the original CP method cannot.

7 Experiments

We have implemented the new procedure in an offline data-race prediction tool based on the LLVM platform. The tool is capable of analyzing traces generated by arbitrary concurrent C/C++ programs written using the POSIX threads. It is worth pointing out that other instrumentation tools, such as the PIN binary instrumentation tool, may also be used to generate the input traces.

We have conducted experiments on two sets of benchmarks. The first set is a collection of traces from small multithreaded programs in the recent literature (e.g. [10,24]). The main purpose is to confirm that our implementation is indeed more accurate than existing methods. For example, the benchmark *cp7* comes from an example used in [24] to illustrate why there is no *CP-predictable* data-race and how CP can avoid false alarms. Our tool discovers that it actually has a real and predictable data-race under the CDSR predictive model. We were surprised initially by the data-race reported by our tool, but confirmed subsequently that this was indeed a real data-race, although it could not be detected by using the CP method.

The second set of benchmarks is a collection of traces from various open-source projects with known bugs, which we use to demonstrate the effectiveness of our method. The set contains bug samples extracted from applications such as Mozilla and MySQL, as well as two open-source projects (`aget-0.4` and `pfscan-1.0`) downloaded from the *sourceforge.net* website. Some of the extracted examples are kindly provided by the authors of [32], whereas others are used in some prior publications (e.g. [15]). All benchmarks are accompanied by test cases to facilitate concrete execution.

We compare the performance of three methods: UCG [10], CP [24], and Poly. All methods were implemented in the same data-race prediction tool to facilitate a fair comparison. (Recall that the original CP algorithm [24] was for Java.) Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory.

Table 1. Comparing our new method with UCG [10] and CP [24] for predicting data-races

Test Program						Partial Trace				Num. Data-races				Time (sec/race)		
name	LOC	thr	lks	cvs	vars	evs	lkevs	coevs	rwevs (r / w)	r-p	r-ucg	r-poly	r-cp	t-ucg	t-new	t-cp
cp1	79	3	1	0	2	21	4	0	6 (2/4)	5	1	0	0	0.021	0.029	0.002
cp2b	113	3	1	0	4	27	4	0	12 (5/7)	11	3	1	1	0.020	0.043	0.005
cp2	113	3	1	0	4	27	4	0	12 (5/7)	11	3	1	1	0.030	0.044	0.002
cp4	67	3	1	0	1	18	4	0	3 (0/3)	3	1	1	1	0.015	0.019	0.002
cp5	168	4	3	0	3	45	14	0	14 (4/10)	12	1	0	0	0.042	0.049	0.006
cp5b	144	4	3	0	3	42	14	0	11 (2/9)	9	1	1	0	0.029	0.035	0.003
cp6	255	5	6	0	5	71	24	0	23 (8/15)	19	1	0	0	0.043	0.059	0.005
cp7	277	5	7	0	6	80	30	0	25 (6/19)	20	1	1	0	0.073	0.080	0.008
cp8	119	3	3	0	2	33	12	0	8 (2/6)	7	1	1	1	0.030	0.035	0.004
vt1	53	2	1	0	1	13	4	0	2 (1/1)	1	1	1	1	0.007	0.009	0.002
vt2	59	2	1	0	2	15	4	0	4 (2/2)	1	1	0	0	0.015	0.021	0.004
vt2b	60	2	1	0	2	15	4	0	4 (2/2)	1	1	1	0	0.008	0.012	0.003
vt2c	56	2	1	0	1	14	4	0	3 (1/2)	1	1	1	0	0.015	0.020	0.005
vt2d	56	2	1	0	1	14	4	0	3 (1/2)	1	1	1	0	0.017	0.023	0.005
vtex4	72	2	1	1	2	19	4	1	6 (2/4)	3	1	0	0	0.016	0.020	0.003
vtex5	73	2	1	1	2	19	4	1	6 (2/4)	3	1	1	0	0.032	0.011	0.005
vtex6	92	3	1	1	2	26	6	2	6 (2/4)	5	0	0	0	0.027	0.008	0.007
Total											**20**	**11**	**5**			
UpdateTimer	266	2	1	0	4	50	32	0	11 (6/5)	6	1	1	1	0.026	0.034	0.006
SeekToItem	246	3	1	0	3	49	28	0	10 (6/4)	7	1	1	1	0.021	0.031	0.004
TimerThread	240	2	2	1	4	51	30	2	10 (5/5)	0	0	0	0	0.020	0.024	0.004
NodeState	176	2	1	0	3	32	20	0	5 (2/3)	1	1	1	1	0.047	0.052	0.004
MysqlLog	181	2	1	0	2	32	20	0	5 (2/3)	2	2	2	2	0.033	0.039	0.004
Loadscript	227	3	2	0	3	50	30	0	8 (4/4)	1	0	0	0	0.048	0.055	0.006
FileTransport	184	2	1	0	2	33	20	0	6 (3/3)	2	2	2	2	0.039	0.048	0.009
CreateThread	178	2	1	0	4	32	16	0	9 (5/4)	3	2	1	1	0.041	0.046	0.003
thrift-1606	44	2	0	0	1	9	0	0	3 (1/2)	2	1	1	1	0.002	0.003	0.000
apache-21285	484	3	1	0	8	69	8	0	50 (35/15)	35	8	2	2	0.019	0.043	0.012
apache-25520	82	3	0	0	2	16	0	0	6 (3/3)	5	3	1	1	0.019	0.033	0.001
maple-cir-list	1393	3	2	0	38	208	56	0	140 (60/80)	43	2	1	1	0.137	0.243	0.047
mysql2011	231	3	2	0	10	53	4	0	37 (18/19)	33	3	3	3	0.011	0.021	0.004
pfscan-1.0-r3	4431	4	4	3	324	791	66	29	678 (288/390)	117	6	3	0	0.007	0.158	0.064
aget-0.4.comb	9277	3	1	0	785	1294	40	0	1243(1089/154)	513	405	11	10	0.132	0.357	0.042
Total											**430**	**30**	**26**			

Table 1 shows the results. In this table, the first six columns show the statistics of the test programs, including the name, the number of lines of code, the number of threads, lock variables, condition variables, and shared variables. The next four columns show the statistics of the input trace, including the number of events (evs), the number of lock events (lkevs), the number of condition variable events (coevs), and the number of read/write events (rwevs). We also provide a break-down of the read and write events.

The next four columns show the statistics of the data-race prediction algorithm. Column *rp* shows the number of *potential data-races*. These are the data-races found by our implementation of a standard lockset based analysis. Column *r-ucg* shows the number of data-races reported by the UCG algorithm. Both lockset and UCG may report spurious data-races but miss no real data-races that are *predictable* from the given traces. Column *r-poly* shows the number of real data-races found by our new algorithm. Column *r-cp* shows the number of data-races found by CP. The last three columns show the runtime performance, which is the average time (seconds) per feasibility check.

In the first set of examples, our new method found more real data-races than CP (11 versus 5), while avoiding all false alarms generated by UCG (a total of 9). In the second set of examples, our new method also found more real data-races than CP (30 versus 26), while avoiding all false alarms generated by UCG (a total of 406).

In terms of runtime performance, our method on average takes longer time than both UCG and CP. This is as expected due to its more involved causality analysis. The benefit of the extra effort is that our method always returns the precise result, without false alarms and missed bugs. Furthermore, the runtime numbers of all three methods are very small – practically negligible – for the targeted application. Therefore, we conclude that our for offline applications, our new method is competitive in that it provides a much more *in-depth* analysis of the concurrent execution traces.

8 Related Work

We have reviewed existing trace-based predictive analysis methods including the lock-set analysis and the lock causality based methods for deciding control state reachability (LAH [9,11], LCG [8], UCG [10], and CP [24]). Our new procedure is more generally applicable and accurate than these existing methods.

Beyond offline bug prediction, there is a large body of work on *online bug detection*, e.g. for detecting data-races [6,1,14] and atomicity violations [3,30]. The distinction between these two types of methods is fairly large. Typically, online methods focus primarily on reducing the runtime overhead, often at the expense of losing precision (false alarms) or decreasing coverage (missed bugs). Whereas offline analysis methods focus primarily on improving the precision and coverage. If spending a few extra seconds or even minutes on analyzing a trace log can lead to the discovery of a few more real bugs, it would considered worthwhile. Our polygraph based method is by far the most accurate method for offline analysis of execution traces with limited observability.

A closely related work is PENELOPE [25,5], which can predict and subsequently confirm atomicity violations and null pointer violations. A core predictive analysis in PENELOPE is the LAH [9] analysis. Since our method improves over LAH, in principle, it may also be used to enhance the analysis in PENELOPE. In addition, PENELOPE confirms the predicted buggy interleavings by trying to re-execute them. This approach works well when it is possible to re-execute in the original environment. Our new method, in contrast, is better suited for applications where re-running the program is impossible, e.g. when the traces are generated on the client site.

Another related work is jPredictor [2], which requires the analysis of a complete execution trace consisting of all global and local instructions involved in the execution. A similar limitation exists in the SAT/SMT based predictive methods [27,29,12,21,23,17,22,28], which require the program source code in conjunction to a trace to construct the prediction model. Although in general, these methods are more powerful, the detailed program code information may not be available in in many application settings. Our new method, in contrast, relies on only a simple trace, which is better suited for applications where detailed information about the program is not available.

9 Conclusions

We have presented a new polygraph based procedure for deciding control state reachability properties in simple execution traces generated by multithreaded programs. We have customized our core analysis procedure for an offline data-race detection tool. In this context, our method is provably more accurate than the existing ones, including the more recent causally-precedes method and the universal causality graph method. We have implemented and evaluated our method through experiments. The results confirm that our procedure is indeed more accurate than the existing ones.

Acknowledgments. We thank the anonymous reviewers for their feedback. This work is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

References

1. Bond, M.D., Coons, K.E., McKinley, K.S.: PACER: proportional detection of data races. In: Programming Language Design and Implementation, pp. 255–268 (2010)
2. Chen, F., Serbanuta, T., Rosu, G.: jPredictor: a predictive runtime analysis tool for java. In: International Conference on Software Engineering, pp. 221–230 (2008)
3. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)
4. Farzan, A., Madhusudan, P., Sorrentino, F.: Meta-analysis for atomicity violations under nested locking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 248–262. Springer, Heidelberg (2009)
5. Farzan, A., Madhusudan, P., Razavi, N., Sorrentino, F.: Predicting null-pointer dereferences in concurrent programs. In: Foundations of Software Engineering, p. 47 (2012)
6. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. Commun. ACM 53(11), 93–101 (2010)
7. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Programming Language Design and Implementation, pp. 293–303 (2008)
8. Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In: International Symposium on Logic in Computer Science, pp. 27–36 (2009)
9. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
10. Kahlon, V., Wang, C.: Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 434–449. Springer, Heidelberg (2010)
11. Kahlon, V., Wang, C.: Lock removal for concurrent trace programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 227–242. Springer, Heidelberg (2012)
12. Kundu, S., Ganai, M.K., Wang, C.: CONTESSA: Concurrency testing augmented with symbolic analysis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 127–131. Springer, Heidelberg (2010)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)

14. Li, D., Srisa-an, W., Dwyer, M.B.: SOS: saving time in dynamic race detection with stationary analysis. In: ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 35–50 (2011)
15. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Architectural Support for Programming Languages and Operating Systems, pp. 37–48 (2006)
16. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
17. Said, M., Wang, C., Yang, Z., Sakallah, K.: Generating data race witnesses by an SMT-based analysis. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 313–327. Springer, Heidelberg (2011)
18. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
19. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: Foundations of Software Engineering, pp. 337–346 (2003)
20. Sen, K., Roşu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 211–226. Springer, Heidelberg (2005)
21. Sinha, A., Malik, S.: Using concurrency to check concurrency: Checking serializability in software transactional memory. In: Parallel and Distributed Processing Symposium (2010)
22. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive analysis for detecting serializability violations through trace segmentation. In: Formal Methods and Models for Codesign, pp. 99–108 (2011)
23. Sinha, N., Wang, C.: On interference abstractions. In: ACM Symposium on Principles of Programming Languages, pp. 423–434 (2011)
24. Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound predictive race detection in polynomial time. In: ACM Symposium on Principles of Programming Languages, pp. 387–400 (2012)
25. Sorrentino, F., Farzan, A., Madhusudan, P.: PENELOPE: weaving threads to expose atomicity violations. In: Foundations of Software Engineering, pp. 37–46 (2010)
26. von Praun, C., Gross, T.R.: Object race detection. In: ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 70–82 (2001)
27. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: Foundations of Software Engineering, pp. 23–32 (2009)
28. Wang, C., Ganai, M.: Predicting concurrency failures in the generalized execution traces of x86 executables. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 4–18. Springer, Heidelberg (2012)
29. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
30. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: International Conference on Software Engineering, pp. 221–230 (2011)
31. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. 32(2), 93–110 (2006)
32. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multiprocessor. In: International Symposium on Computer Architecture, pp. 325–336 (2009)

Modular Synthesis of Sketches Using Models

Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama[⋆]

Massachusetts Institute of Technology

Abstract. One problem with the constraint-based approaches to synthesis that have become popular over the last few years is that they only scale to relatively small routines, on the order of a few dozen lines of code. This paper presents a mechanism for modular reasoning that allows us to break larger synthesis problems into small manageable pieces. The approach builds on previous work in the verification community of using high-level specifications and partially interpreted functions (we call them models) in place of more complex pieces of code in order to make the analysis modular.

The main contribution of this paper is to show how to combine these techniques with the counterexample guided synthesis approaches used to efficiently solve synthesis problems. Specifically, we show two new algorithms; one to efficiently synthesize functions that use models, and another one to synthesize functions while ensuring that the behavior of the resulting function will be in the set of behaviors allowed by the model. We have implemented our approach on top of the open-source Sketch synthesis system, and we demonstrate its effectiveness on several Sketch benchmark problems.

1 Introduction

Over the last few years, constraint-based approaches to synthesis based on sketches or templates have become quite popular [11,19,26,28,29]. In these approaches, the user specifies her intent with a template of the desired solution leaving parts of the code unspecified (a sketch), and the synthesizer finds the unknown code fragments such that the completed template conforms to a given set of behavioral constraints (the spec). One problem with such approaches, however, is that they only scale to relatively simple routines, on the order of a few dozen lines of code. Scaling such methods to more complex programs requires a mechanism for modular reasoning.

The idea of modular reasoning has been quite successful and is widely used today in verification of software and hardware, where pre and post-conditions are commonly used to model complex functions (*e.g.* MAGIC [7],DAFNY [14]), and where uninterpreted or partially interpreted functions play an important role in abstracting away complex functional units [4,5]. These assume-guarantee

⋆ This work was funded by NSF grants NSF-1116362, NSF-1139056, and DOE grant DE-SC0005372.

reasoning based approaches perform compositional verification by breaking down the verification task of a system into smaller tasks that involve verification of individual components, which enables the verification tools to then compose the proofs to verify the whole system [18,30].

In this paper, we present a mechanism of modular reasoning for synthesizing complex sketches, where we use *function models* to specify the behavior of constituent function components. A function model consists of three components: pre-processing code that canonicalizes the input, an uninterpreted function that models the function behavior, and a post-condition that specifies the desired properties of a function. A model can also have a pre-condition that specifies which parameters are legal for the function. However, we place two restriction on models: 1) they cannot have any unknown code fragment (holes) and 2) they cannot post-process the output of the uninterpreted function before returning it. These function models are more general than pre and post-conditions without uninterpreted functions, but are less general than pre and post-conditions with quantifiers. This intermediate generality of function models provides us the expressiveness to specify many complex functions and at the same time allows us to efficiently use them for synthesis.

The function models in sketches introduce two new synthesis problems: 1) synthesis with models and 2) synthesis for adherence. The synthesis with models problem requires us to solve for unknown control parameters such that for any uninterpreted function that satisfies the model's post-condition, the specification of main function should also hold. This problem in principle can be solved by the traditional Cegis [24] algorithm but Cegis performs poorly in practice. We present a new algorithm Cegis+ that combines the Cegis algorithm with an approach that selectively uses existential quantification for some inputs [11] for efficiently solving this problem. The sketch will use the function model in place of a more complex function that itself may have unknowns. Therefore, we also need to solve a second synthesis problem: synthesis for adherence, which ensures that this more complex function is synthesized in a way that matches the behavior promised by the model. This synthesis problem introduces a standard doubly quantified constraint but with an existentially quantified function to find a function that satisfies the model post-condition on all valid inputs and has an equivalent input-output relationship with the original function. To solve this problem, we present an approach to eliminate the existentially quantified function by taking advantage of the specific form of the constraint and do not require additional function templates unlike previous approaches [24,33].

We have implemented the algorithms in the Sketch synthesis system [25] and present the evaluation of the algorithm on several benchmark problems. We show that function models enable synthesis of several complex benchmarks, and our algorithm outperforms both Cegis and previously published algorithms.

Specifically, the paper makes the following contributions:

– We show that the existing approaches to counterexample guided synthesis break down in the presence of models and present a new algorithm that can efficiently synthesize functions that use models.

- We present a new way to encode the problem of synthesizing a function that behaves according to a model without the need for existentially quantified functions.
- We present the evaluation of our algorithm on several benchmark problems and show how it enables solving of complex sketches.

2 Motivating Examples

We use two running examples to motivate the need of function models: integer square root and big integer multiplication. They represent different kinds of models, namely fully specified models and partially specified models, which will expose different aspects of our algorithm.

2.1 Example 1: Square Root for Primality Testing

For the first example, consider the sketch of an algorithm for primality testing shown in Figure 1(a). The sketch requires the synthesizer to discover many of the details of a fast primality testing algorithm that computes much smaller number of divisibility checks than $\sqrt{\mathtt{p}}$ for every input `p`. Most importantly for our purposes, the sketch calls a `sqrt` function to compute the integer square root. This `sqrt` function is also sketched—not shown in the figure—so the synthesizer would normally have to derive the details of the primality test and of the `sqrt` function simultaneously. The correctness of the resulting implementation will be established by comparing the result to that of a linear time primality test, one that simply tries to divide `p` against all integers less than `p`. Sketch uses bounded reasoning when deriving the details; in the case of this example, it will only consider values of `p` with up to 8 bits. The sketch uses the `optimize` function to ensure that the bound `bnd` is minimized for any value of `p`. The `linexp` construct is not a function but a generator that must be replaced by the synthesizer with a linear expression over its arguments and the `minrepeat` construct repeats its argument statement minimum number of times (with different holes) such that the sketch becomes satisfiable.

Instead of solving for the primality test and the square root simultaneously, we can write a model of the `sqrt` function to express its main properties. In general, a model calls an uninterpreted function and then asserts some properties of the return value(s), which corresponds to the post-conditions of the modeled function. The model allows us to break the problem into two sub-problems: 1) we need to solve the main sketch using the model in place of the more complex square root function, 2) we need to solve for the details of the square root function under the constraint that it behaves according to the model. Note that in addition to establishing the post-conditions, the model also expresses the fact that when called twice with the same input, `msqrt` will produce the same value. For this example the property is not very important because the function `sqrt` is only invoked once. The post-condition fully constrains the output for any input, so the model is said to be fully specified.

```
harness void fastPrimalityCheck(int p){
  // all primes are of form 6k±1 except 2,3
  bit isPrime = false;
  if(p>??){
    isPrime = true;
    // repeat minimally with different holes
    // check divisibility by 2 or 3
    minrepeat{if(p%?? == 0) isPrime = false;}
    // minimize loop bound: l1(√l2(p))/n
    int bnd = linexp(sqrt(linexp(p))) / ??;
    optimize(bnd,p);
    for(int i=??; i < bnd; ++i){
      minrepeat{
        if(p % linexp(i) == 0) isPrime = false;}
    }
  }
  assert isPrime==checkPrimalityLinear(p);
}
```

(a)

```
int msqrt(int i) models sqrt{
    int rv = sqrtuf(i);
    if(i<=0){
        assert rv == 0;
    }else{
       assert rv*rv <= i;
       assert (rv+1)*(rv+1)>i;
    }
    return rv;
}
```

(b)

Fig. 1. (a) A sketch harness using the `sqrt` function to find the fast primality check algorithm(en.wikipedia.org/wiki/Primality_test) that requires much lesser than $\sqrt{p}$ divisibility checks, (b) a model for the `sqrt` function encoding the square root property

2.2 Example 2: Big Integer Multiplication

Models don't have to be fully specified; in many cases, only a handful of properties of a function are relevant to synthesize a piece of code. As an example, consider an application that requires big-integer multiplication; Section 5 describes a couple of such functions we have explored, one of which is taking derivatives of polynomials with big-integer coefficients. For all of these experiments, we were interested in synthesizing implementations that used the Karatsuba algorithm for big-integer multiplication, whose details we also wanted to synthesize as was done in [25].

Without models, solving the sketch requires reasoning about the main function and the big integer multiplication in tandem. However, to reason about the polynomial derivative function, we only need to know that multiplication is commutative, and the zero property of multiplication (a number times zero is equal to zero). We describe those properties in the model in Figure 2(b). Just like with `sqrt`, the model breaks the problem into two independent sub-problems, but there are important differences between the `msqrt` model and the `mmul` model. The `mmul` model is under-specified, so there may be many functions that satisfy it. This means that when synthesizing karatsuba, the constraint that the solution is represented by the model must be combined with additional constraints that ensure that it is indeed implementing big-integer multiplication. Also, the model uses `min` and `max` to canonicalize the input so that `mmul(a,b)` will call the

```
harness void main(int[n] x1, int[n] x2){
  ...
  t = mul(x1, x2);
}

int[n] mul(int[n] x1, int[n] x2){
  // karatsuba algorithm
    ...
}
```

(a)

```
int[n] mmul(int[n] x1, int[n] x2)
                        models mul{
    int[n] xa = min(x1, x2);
    int[n] xb = max(x1, x2);
    int[n] rv = muluf(xa, xb);
    if(x1 == 0 || x2==0){
      assert rv == 0;
        }
        return rv;
}
```

(b)

Fig. 2. (a) A sketch harness using the `mul` function that uses the karatsuba algorithm for multiplying two integers represented by integer arrays, and (b) a model for the `mul` function encoding the commutativity and zero properties

uninterpreted function with the same arguments as `mmul(b,a)` and therefore the model will be commutative.

Both the multiplication and the square root model use the same basic mechanisms, but as we will see they interact very differently with the counterexample guided inductive synthesis algorithm Cegis. One of the challenges addressed by our work is to define new general algorithms that work efficiently for both fully specified and partially specified models with combinations of interpreted and uninterpreted functions.

3 Problem Definition

A sketch is a function with missing code fragments marked by placeholders; however, all sketches can be represented as parametrized functions $Sk[c](in)$ or simply $Sk(c, in)$, where the parameter c controls the choice of code fragments to use in place of the unknowns. The role of synthesizer is therefore to find a value of c such that for all inputs $in \in E$ in a given input space E, the assertions in the sketch will be satisfied. In some cases, we may also want to assert that the sketch is functionally equivalent to a separately provided specification. This definition comes from [27], and using this definition as a starting point, we can formalize our support for function models.

Figure 3(a) shows a canonical representation of the models supported by our system. A model $\mathcal{M}$ is defined to be a 3-tuple $\mathcal{M} \equiv (\alpha, f_u, \mathsf{P}_{\mathsf{model}})$, where α denotes the canonicalization function that canonicalizes the input to the model before passing it to the uninterpreted function, f_u denotes an uninterpreted function whose output `rv` is the output of the model, and $\mathsf{P}_{\mathsf{model}}(\mathsf{rv}, \mathsf{in}_{\mathsf{model}})$ denotes a predicate that establishes properties of the return value with respect to the input. The predicate $\mathsf{P}_{\mathsf{model}}$ encodes the function's post-condition. We assume that models do not have explicit preconditions as they can be added using additional if statements inside the $\mathsf{P}_{\mathsf{model}}$ predicate, and they do not add much to the formalism.

```
FModel(in_model) models f_orig{
  /* canonicalization function */
  x = α(in_model);
  /* uninterpreted function */
  rv = f_u(x);
  /* post-condition */
  assert P_model(rv, in_model);
  return rv;
}
```

(a)

```
harness void Main(in){
  c_1 = ??; // unknown control
  /* arbitrary computation */
  t_1 = h(in, c);
  /* original function call */
  t_2 = f_orig(t_1);
  /* sketch assertion */
  assert P_main(t_2, in, c);
}
  f_orig(in){
     c_2 = ??;
     ...
  }
```

(b)

Fig. 3. (a) A simple canonical model for a function $f_{\mathtt{orig}}$, and (b) a sketch function `Main` using the function $f_{\mathtt{orig}}$

Example 1. For the `sqrt` function model in Figure 1, the uninterpreted function f_u is `sqrtuf`, the canonicalization function is the identity function $\alpha(i) = i$, and the predicate is $\mathtt{P_{model}} \equiv (i \leq 0 \rightarrow \mathtt{rv} = 0) \wedge (i > 0 \rightarrow \mathtt{rv}^2 \leq i \wedge (\mathtt{rv}+1)^2 > i)$. For the big integer multiplication model in Figure 2, the uninterpreted function f_u is `muluf`, the canonicalization function is $\alpha(x_1, x_2) = (\mathtt{min}(x_1, x_2), \mathtt{max}(x_1, x_2))$, and the predicate is $\mathtt{P_{model}} \equiv (x_1 = 0 \vee x_2 = 0) \rightarrow rv = 0$.

We now formalize the problem in terms of a stylized sketch shown in Figure 3(b) which uses a function $f_{\mathtt{orig}}$ for which a model will be provided. In general, sketches can have a large number of unknowns, but in our stylized function, we use the variable c to represent the set of unknown values that the synthesizer must discover. The function $h(\mathtt{in}, c)$ represents an arbitrary computation on the inputs to the main function to generate the inputs to the model. The unknown values may flow to the $f_{\mathtt{orig}}$ function, but as far as that function is concerned, they are just another input. The function $f_{\mathtt{orig}}$ itself may have additional unknowns, but the model cannot. In the constraint formulas, we will use $f_{\mathtt{orig}}(\mathtt{in}, c)$ to denote the fact that the unknown values in $f_{\mathtt{orig}}$ will also be discovered by the synthesizer. The correctness of the overall function is represented by a set of assertions which can be represented by a predicate $\mathtt{P_{main}}(t_2, \mathtt{in}, c)$, which can be expanded to $\mathtt{P_{main}}(f_{\mathtt{orig}}(h(\mathtt{in}, c), c), \mathtt{in}, c)$. Without loss of generality, we assume that the main function includes a single call to $f_{\mathtt{orig}}$, but our actual implementation supports multiple calls to $f_{\mathtt{orig}}$. Also, in real sketches, the assertions can be scattered throughout the function, but this stylized sketch function will illustrate all the key issues in supporting models.

Now, to compute the value of unknown parameter c from the sketch, we need to solve the following two constraints.

1. Correctness of main under the model *(Correctness constraint)*

$$\exists c_1 \forall \mathtt{in} \forall f_u \ \mathsf{P}_{\mathsf{model}}(\mathsf{rv}, \mathtt{in}_{\mathsf{model}}) \rightarrow \mathsf{P}_{\mathsf{main}}(\mathsf{rv}, \mathtt{in}, c_1) \tag{1}$$

   where $\mathtt{in}_{\mathsf{model}} \equiv h(\mathtt{in}, c_1)$ and $\mathsf{rv} \equiv f_u(\alpha(\mathtt{in}_{\mathsf{model}}))$. The constraint establishes that we want to find unknowns c_1 to complete the sketch such that for any function f_u, if the function satisfies the assertions in the model, on a given input, then the assertions inside main will also be satisfied.
2. Adherence of the original function to the model *(Adherence constraint)*

$$\exists c_2 \exists f_u \forall x \ \mathsf{P}_{\mathsf{model}}(f_u(\alpha(x)), x) \wedge f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c_2) \tag{2}$$

   The constraint establishes that there exists a function f_u that satisfies the assertions on all valid inputs $(\alpha(x))$, and that has an equivalent input-output relationship with the original function f_{orig}.

As the following theorem explains, finding c_1 and c_2 that satisfy the two constraints above is equivalent to finding a solution to the original sketch problem.

Theorem 1. *If both Correctness constraint (Eq. 1) and Adherence constraint (Eq. 2) are satisfied, then*

$$\exists c_1, c_2 . \forall in . P_{main}(f_{orig}(h(in, c_1), c_2), in, c_1)$$

Proof. Since the Adherence constraint (Eq. 2) is satisfied, we can use the second conjunct $f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c_2)$ to substitute $f_u(\alpha(x))$ with $f_{\mathsf{orig}}(x, c_2)$ in the first conjunct of Eq. 2 to obtain

$$\exists c_2 \forall x \ \mathsf{P}_{\mathsf{model}}(f_{\mathsf{orig}}(x, c_2), x) \tag{3}$$

Since the Correctness constraint (Eq. 1) holds for all f_u, it should hold for the f_u in the solution of the Adherence constraint, and therefore $f_u(\alpha(\mathtt{in}_{\mathsf{model}}))$ can be substituted with $f_{\mathsf{orig}}(\mathtt{in}_{\mathsf{model}}, c_2)$ in Eq. 1 to obtain:

$$\exists c_1, c_2 \forall \mathtt{in} \ \mathsf{P}_{\mathsf{model}}(f_{\mathsf{orig}}(\mathtt{in}_{\mathsf{model}}, c_2), \mathtt{in}_{\mathsf{model}}) \rightarrow \mathsf{P}_{\mathsf{main}}(f_{\mathsf{orig}}(\mathtt{in}_{\mathsf{model}}, c_2), \mathtt{in}, c_1) \tag{4}$$

In the above constraint (Eq. 4), we know the left hand side of implication holds from Eq. 3, therefore the right hand side of implication should also hold, i.e. $\exists c_1, c_2 \forall \mathtt{in} \ \mathsf{P}_{\mathsf{main}}(f_{\mathsf{orig}}(\mathtt{in}_{\mathsf{model}}, c_2), \mathtt{in}, c_1)$ holds where $\mathtt{in}_{\mathsf{model}} \equiv h(\mathtt{in}, c_1)$.

4 Solving Correctness and Adherence Constraints

In previous work [24], we have used a counterexample guided inductive synthesis (Cegis) approach to solve the doubly quantified constraints that arise in synthesis. Given a constraint of the form $\exists c. \ \forall \mathtt{in}. \ \mathtt{Q}(\mathtt{in}, c)$, Cegis solves an inductive synthesis problem of the form $\exists c. \ \mathtt{Q}(\mathtt{in}_0, c) \wedge \mathtt{Q}(\mathtt{in}_1, c) \cdots \wedge \mathtt{Q}(\mathtt{in}_k, c)$, where $\{\mathtt{in}_0 \cdots, \mathtt{in}_k\}$ is a small set of representative inputs. If the equation above is unsatisfiable, the original equation will be unsatisfiable as well. If the equation provides a solution, we can check that solution by solving the following equation $\exists \texttt{in}\ \neg\texttt{Q}(\texttt{in}, c)$. The algorithm, shown in Figure 4, consists of two phases: synthesis phase and verification phase. The algorithm first starts with a random assignment of inputs $\texttt{in}_0$ and solves for the constraint $\exists c\ \texttt{Q}(\texttt{in}_0, c)$. If no solution exists, then it reports that the sketch can not be synthesized. Otherwise, it passes on the solution c to the verification phase to check if the solution works for all inputs using the constraint $\exists \texttt{in}\ \neg\texttt{Q}(\texttt{in}, c)$. If the verifier can't find a counterexample input, then the sketch $Sk(c)$ is returned as the desired solution. Otherwise, the verifier finds a counterexample input $\texttt{in}_1$ which is then added to the synthesis phase. The synthesis phase now solves for the constraint $\exists c\ \texttt{Q}(\texttt{in}_0, c) \wedge \texttt{Q}(\texttt{in}_1, c)$. This loop between the synthesis and verification phases continues until either the synthesis or the verification constraint becomes unsatisfiable. The algorithm returns "no solution" when the synthesis constraint becomes unsatisfiable whereas it returns the sketch solution when the verification constraint becomes unsatisfiable.

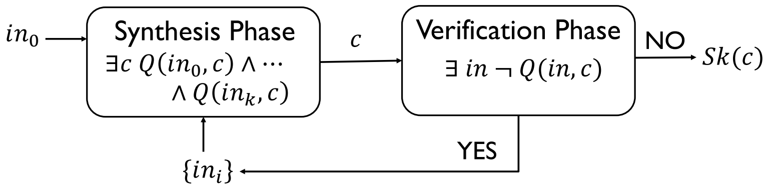

in_0
Synthesis Phase
$\exists c\ Q(in_0, c) \wedge \cdots \wedge Q(in_k, c)$
c
Verification Phase
$\exists\ in\ \neg\ Q(in, c)$
NO
$Sk(c)$
YES
$\{in_i\}$

Fig. 4. The CounterExample Guided Inductive Synthesis Algorithm (CEGIS)

4.1 Limitations of CEGIS for the Correctness constraint

We can apply the same CEGIS approach to solve the Correctness constraint in Eq. 1, but as several authors [11,33] have pointed out, the CEGIS algorithm tends to perform poorly when there are strong assumptions in the sketch that depend on the values of the unknown control parameters. The problem is that when the verifier finds a counterexample, it is relatively easy for the inductive synthesizer to avoid the problem by changing the assumptions rather than by correcting the problem. To illustrate this issue, consider the example in Figure 5.

```
harness void main(in){
    int j = in + c1;
    int t = msqrt(j*j);
    assert t == in + c2;
}
```

Fig. 5. A simple msqrt example

The example is artificial, but it illustrates an effect that happens in the primality check example as well. The example has two unknown integer values, c_1 and c_2, and one can easily see that as long as c_1 and c_2 are equal, the program will be correct. In this case, a counterexample from the verifier would include the value of `in`, as well as a function `sqrtuf`. Now, suppose that the CEGIS algorithm starts with an initial guess of $c_1 = 3$ and $c_2 = 6$. The verifier can immediately produce the following counterexample: $\texttt{in} = 2, \texttt{sqrtuf} = (25 \rightarrow 5, \texttt{else} \rightarrow 7)$. The function `sqrtuf` in the counterexample evaluates to 5 when the input is 25, and to 7 otherwise. In this case, the strong sketch assumption (the model assertion) $t{*}t \leq (in{+}c_1){*}(in{+}c_1) \wedge (t{+}1){*}(t{+}1) > (in{+}c_1){*}(in{+}c_1)$ depends on the value of the control parameter c_1. The problem now is that for any value of $c_1 \neq 3$, the `sqrtuf` function in the counterexample will fail the model assertions. Say the synthesizer picked a value $c_1 = 4$, then the model assertion $\mathsf{P_{model}} \equiv 7 * 7 \leq (2+4) * (2+4)$ becomes false. The synthesizer can easily pick a value for $c_1 \neq 3$ that makes the model assertions $\mathsf{P_{model}}$ false, and therefore vacuously satisfy the correctness condition $\mathsf{P_{model}} \rightarrow \mathsf{P_{main}}$. Therefore, the CEGIS loop needs to perform $O(2^n)$ iterations before converging to the desired solution, where n is bound on the number of bits in the input integer `in`.

A previously proposed solution to this problem has been to identify that some inputs are actually dependent on other inputs, and should therefore not be a part of the counterexample, but instead the values of the dependent inputs should be existentially quantified [11]; *i.e.* they should be chosen *angelically* to use the terminology of [3,6]. In this case, for example, that approach would suggest that the `sqrtuf` function should not be part of the counterexample, since it is fully determined by the assertions in the model and the values of `in` and c_1. Following this approach, the inductive synthesis problem would then be

$$\exists c, f_{u_0}, \cdots, f_{u_k}.\ \mathsf{Q}(c, \mathsf{in}_0, f_{u_0}) \wedge \cdots \wedge \mathsf{Q}(c, \mathsf{in}_k, f_{u_k}) \tag{5}$$

where $\mathsf{Q}(c, \mathsf{in}, f_u) \equiv \mathsf{P_{model}}(f_u(\alpha(h(\mathsf{in}, c))), h(\mathsf{in}, c)) \wedge \mathsf{P_{main}}(f_u(\alpha(h(\mathsf{in}, c))), \mathsf{in}, c)$. Note that here we have replaced the implication in the correctness constraint with the conjunction, enforcing that the angelically selected f_{u_i} values always satisfy the model assertions $\mathsf{P_{model}}$. The function f_u is no longer part of the counterexample, since it is fully defined by the assertions from the values of input `in` and unknown control c. This approach has an implicit assumption that there exist functions f_{u_i} that satisfies $\mathsf{P_{model}}$ for corresponding inputs.

However, a big problem with this approach is that it may fail to converge in some cases. This happens in cases when the predicates in the model do not fully constrain the function, or they constrain it fully on only some of the inputs. For example, this is the case with the big integer multiplication example, where the predicate only constrains the function when one of the inputs is zero. Consider the scenario where for a given value of unknown c_i and an input in_i, there are two functions f_u and f'_u that both satisfy the model assertions $\mathsf{P_{model}}$, but only f_u satisfies the `main` assertions $\mathsf{P_{main}}$, i.e. $\mathsf{Q}(c_i, \mathsf{in}_i, f_u)$ is satisfiable whereas $\mathsf{Q}(c_i, \mathsf{in}_i, f'_u)$ is unsatisfiable. Since the synthesizer is solving the existential (angelic) problem in Eq. 5, it will satisfy the equation by selecting the function f_u for in_i and c_i. However, the verifier needs to ensure that the completed sketch satisfies the correctness predicate for all functions f_u, and it will produce a counterexample (in_i, f'_u) for the given control value c_i. Since the synthesizer ignores the function f'_u of the counterexample and only considers the input in_i, this algorithm as a result goes into an infinite loop and does not converge.

4.2 Our Algorithm Cegis+

Our algorithm Cegis+ combines the benefits of the angelic approach while also ensuring that it converges in all cases. For the inductive synthesis phase, we use the following constraint:

Definition 1 (Inductive synthesis constraint)

$$\exists c, f_{u_0}, \cdots, f_{u_k} \bigwedge_{(\mathit{in}_i, f_{u_i}^{\mathsf{cex}})} \begin{array}{c} \mathit{let}\ y = h(\mathit{in}_i, c), x = \alpha(y)\ \mathit{in} \\ \mathit{let}\ t = \mathit{ite}(P_{\mathit{model}}(f_{u_i}^{\mathsf{cex}}(x), y), f_{u_i}^{\mathsf{cex}}(x), f_{u_i}(x))\ \mathit{in} \\ P_{\mathit{model}}(t, y)\ \wedge\ P_{\mathit{main}}(t, \mathit{in}_i, c) \end{array}$$

where $\mathtt{ite}(c, a, b)$ *is the standard if-then-else function such that* $\mathtt{ite}(\mathtt{true}, a, b) = a$ *and* $\mathtt{ite}(\mathtt{false}, a, b) = b$. *The functions* $f_{u_i}^{\mathsf{cex}}$ *are obtained from the verifier counterexamples, and the functions* f_{u_i} *are determined angelically.*

The key idea behind this approach is that if the function from the counterexample satisfies the model assertions in the synthesis phase, then the synthesis constraint will use the counterexample function in the model. This will often be the case when the assertions in the model are under-constrained (weak), as is the case in the big integer multiplication example. On the other hand, if the assertions in the model are strong, as is the case with the square root model, the counterexample function will be ignored, and instead the synthesizer will use one of the angelically determined functions. The verification phase still solves the same correctness constraint in Eq. 1. The soundness of our algorithm follows from the soundness of the Cegis and angelic algorithms. We now show that Cegis+ algorithm always converges.

Theorem 2. *Assuming there exists a function* f_u *that satisfies the model assertions for all inputs, the* Cegis+ *algorithm is guaranteed to converge to the solution of correctness constraint in Eq. 1.*

Proof. Since all sketches are solved with a bounded size on inputs, the set of possible counterexamples $(\mathtt{in}, f_u^{\mathsf{cex}}) \in \mathcal{IN} \times \mathcal{F_U}$ is bounded where input in and function f_u^{cex} take values from the finite sets $\mathcal{IN}$ and $\mathcal{F_U}$ respectively. In each iteration of the Cegis+ algorithm a new counterexample $(\mathtt{in}, f_u^{\mathsf{cex}})$ is added. The only case for the algorithm to iterate forever is when the inductive synthesizer can produce a c that fails for one of the previously found counterexamples $(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})$ (i.e. it ignores the $f_{u_i}^{\mathsf{cex}}$ value and selects the angelic value f_{u_i} instead) and the verifier generates the counterexample $(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})$. This can't happen, because our synthesis constraint only ignores the counterexample function when the model assertion $\mathsf{P}_{\mathsf{model}}(f_{u_i}^{\mathsf{cex}}(x), y)$ becomes false and therefore $(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})$ cannot be a valid counterexample as the implication $\mathsf{P}_{\mathsf{model}} \rightarrow \mathsf{P}_{\mathsf{main}}$ will be true vacuously.

4.3 Solving the Adherence Constraint

Once we know that the main function is correct under the model, we need to show that the original function actually matches the behavior promised by the model. The adherence of model to the original function can be established by the following constraint:

$$\exists c\ \exists f_u\ \forall x.\ \mathsf{P}_{\mathsf{model}}(f_u(\alpha(x)), x) \wedge f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c) \tag{6}$$

This constraint looks similar to the standard doubly quantified constraint usually solved by CEGIS, but one crucial difference is that it contains an existentially quantified function. The Z3 SMT solver [33] uses two main approaches to get rid of these kinds of uninterpreted functions; one is to treat assignments of the form $\forall x.f(x) = t[x]$ as macros and rewrite all occurrences of $f(x)$ to $t[x]$ in the formula. We can use this technique to eliminate f_u from the first part of the equation above, but the equality $f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c)$ cannot in general be treated as a macro because of the presence of α. Another approach used by Z3 and inspired by Sketch is to ask the user to provide a template for f_u that allows it to do existential quantification exclusively over values instead of over functions. However, in our case we can do a lot better than that by taking advantage of the specific form of the constraints and the fact that we do not actually care about what f_u is; we only care to know that it exists.

We can efficiently solve the Adherence constraint in Eq. 6 by instead solving the following equivalent constraint:

$$\exists c \forall x\ \mathsf{P}_{\mathsf{model}}(f_{\mathsf{orig}}(x, c), x)\ \wedge\ \forall x_1, x_2\ \alpha(x_1) = \alpha(x_2) \rightarrow f_{\mathsf{orig}}(x_1, c) = f_{\mathsf{orig}}(x_2, c) \tag{7}$$

The constraint states that the original function should satisfy the model constraints for all input values x, and if two inputs x_1 and x_2 cannot be distinguished by the canonicalization function α then the original function f_{orig} should also produce the same outputs on the two inputs. This equation does not involve any uninterpreted functions of any kind, and can be solved efficiently by the standard CEGIS algorithm because the left hand side of the implication does not depend on the unknown values c.

Theorem 3. *The constraint in Eq. 7 is equivalent to the Adherence constraint in Eq. 6.*

Proof. It is easy to see that the Adherence constraint (Eq. 6) implies the constraint in Eq. 7. If f_{orig} does not satisfy the assertions in the model, then it can not be equal to $f_u(\alpha(x))$. Also, if there are two inputs that cannot be distinguished by α, but for which f_{orig} produces different outputs, then it would not be possible to find an f_u such that $f_u(\alpha(x))$ equals f_{orig}.

The converse is a little trickier. We have to show that if Eq. 7 is satisfied, then the Adherence constraint will be satisfied as well. The key is to show that for a given value of c if $\forall x_1, x_2.\ \alpha(x_1) = \alpha(x_2) \rightarrow f_{\mathsf{orig}}(x_1, c) = f_{\mathsf{orig}}(x_2, c)$, then $\exists f_u \forall x\ f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c)$. Let $f_u(t)$ be a function computed as follows:

$$f_u(t) = \begin{cases} f_{\mathsf{orig}}(x_1, c) & \text{if } \exists x_1.\ \alpha(x_1) = t \\ 0 & \text{Otherwise} \end{cases}$$

Now, we have to show that such an f_u is well defined and satisfies $\forall x f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c)$. Consider two values x_1 and x_2 such that $\alpha(x_1) = \alpha(x_2) = t$, then $\alpha(x_1) = \alpha(x_2) \rightarrow f_{\mathsf{orig}}(x_1, c) = f_{\mathsf{orig}}(x_2, c)$ gives us $f_{\mathsf{orig}}(x_1, c) = f_{\mathsf{orig}}(x_2, c)$. So the function $f_u(t)$ returns the same value for both x_1 and x_2 and is therefore well defined. The function f_u satisfies the constraint $\forall x f_u(\alpha(x)) = f_{\mathsf{orig}}(x, c)$ by definition.

A final point to note is that if a function f_{orig} satisfies the Adherence constraint for a given model, then it must be true that there exists an f_u such that the model satisfies its assertions for all inputs, which was one of the assumptions of the algorithm to solve the Correctness constraint.

5 Evaluation

We now present the evaluation of our algorithms on a set of Sketch benchmark problems. All these benchmark problems consists of sketches that use complex functions such as integer square root, big integer multiplication, sorting (array, topological) etc. In our evaluation, we run each benchmark problem for 20 runs and we present the median values for the running times and the number of iterations of the synthesis-verification loop. The experiments were run (for parallelization) on virtual machines with physical cores using Intel Xeon L5640 2.27GHz processors, each virtual machine comprising of 4 virtual CPUs (2 physical cores) and 16 GB of RAM.

5.1 Implementation and Benchmarks

We have implemented our algorithms for solving the Correctness and Adherence constraints on top of the open-source Sketch solver. Our benchmark problems can be found on the Sketch server[1]. A brief description of the set of sketch benchmarks that we use for our evaluation is given below.

- `calc-toposort`: A function for evaluating a Boolean DAG using topological sort function. A more detailed case study is presented in Section 6 for this benchmark.
- `bsearch-sort`: A binary search algorithm to find an element in an array that uses the sort function.
- `gcd-n-nums`: An algorithm to compute the gcd of n numbers that uses the gcd function.
- `lcm-n-nums`: An algorithm to compute the lcm of n numbers that uses the lcm function.
- `matrix-exp`: An algorithm to compute matrix exponentiation using the matrix multiplication function.

[1] `http://sketch1.csail.mit.edu/Dropbox/models/experiments/`

- `polyderiv-mult`: An algorithm to compute the derivative of a polynomial whose coefficients are represented using big integer representation and that uses karatsuba multiplication.
- `polyeval-mult-exp`: An algorithm to compute the value of a polynomial on a given value that uses the karatsuba multiplication and exponentiation functions.
- `power-root-sqrt`: An algorithm to compute the 2^kth integer root of a number using the integer square root function.
- `primality-sqrt`: An algorithm to check if a number is prime that uses the integer square root function.

Experimental Setup. All our benchmarks include a larger `main` function sketch which calls another function $f_{\mathtt{orig}}$ which we would like to model and perform modular synthesis efficiently. In most of the cases, the inner function $f_{\mathtt{orig}}$ is a sketch which comes with an imperative specification `f-spec` and a declarative model `f-model`. We perform our experiments based on the strength of the models:

1. If `f-model` enforces strong constraints (fully specifying the function, e.g. the `sqrt` model) then we use `f-model` to synthesize both `main` and $f_{\mathtt{orig}}$. We compare this with synthesis of $f_{\mathtt{orig}}$ with the imperative `f-spec` and then using `f-spec` or the synthesized $f_{\mathtt{orig}}$ function to synthesize `main`.
2. If `f-model` enforces weak constraints (partially specifying the function e.g. the `mult` model) then we will have to fallback to synthesizing $f_{\mathtt{orig}}$ using `f-spec` in any case. So, we don't show the time for this synthesis process and simply compare the median times for synthesis of `main` using `f-model`, `f-spec` or synthesized $f_{\mathtt{orig}}$.

5.2 Scaling Sketch Solving Using Models

We first show the need of using function models for solving large complex sketches. The last columns in Table 1 and Table 2 show the time required by the Sketch solver to synthesize the `main` function using the synthesized code for inner function $f_{\mathtt{orig}}$, which involves solving two sketch harnesses. As we can see from the tables, most of these sketches either timeout because of overshooting the memory limits or by going over the timeout limit, which we set to 15 minutes for all the benchmark runs except the `calc-toposort` benchmark for which we use a 5 hour limit. We observed that even when we let these sketches run for a longer time, they often run out of memory and do not terminate. Other alternative options to solve such complex sketch problems is to synthesize the function $f_{\mathtt{orig}}$ using its imperative specification `f-spec`, and then synthesize the harness function using `f-spec`. The middle column(s) in Table 1 and Table 2 (`f-spec`) report the time taken to synthesize the `main` function using `f-spec`. We observe that this cleaner separation allows some of the harness functions to be synthesized, but it typically takes a very long time. Some of these benchmarks do not terminate when we use more complex functions, e.g. when we use merge sort (instead of bubble sort) for the sorting function. The first columns in Table 1 and

Table 2 (Using `f-model`) show the results of using function models for solving `main` or both of these sketches. We observe a big improvement in synthesis times of sketches for cases in which they depend on only some partial property of $f_{\texttt{orig}}$ and in cases where the models are exponentially succinct. For example, for the `matrix-exp` benchmark, the sketch harness only needs to know the exponentiation property of multiplication. For some benchmarks such as `primality-sqrt`, the synthesis times are quite similar to the synthesis time of the second approach because in this case the function model expresses the complete property of the sqrt function, and the constraints generated by the model are almost equal in size to the constraints generated by the linear square root search. We note that in all the benchmarks, the model based solving is always faster than the chained synthesis (Synthesizing `main` using synthesized $f_{\texttt{orig}}$) and in most cases, it is also better than or as good as using the imperative specification `f-spec`.

Table 1. The sketch solving times for three approaches in the presence of a strong model: i) using models (`f-model`) ii) using imperative specification `f-spec`, and iii) Synthesis of $f_{\texttt{orig}}$ using `f-spec` and `main` using synthesized $f_{\texttt{orig}}$. The × values in the table entries denote timeout (> 15 mins), *timeout for calc-toposort set to 5 hours.

Benchmark	Solving Time (in s) for Synthesis of `main`									
	Using `f-model`				Using `f-spec`			Using synthesized $f_{\texttt{orig}}$		
	`main`	$f_{\texttt{orig}}$	Adh.	Total	`main`	$f_{\texttt{orig}}$	Total	`main`	$f_{\texttt{orig}}$	Total
`calc-toposort`	246.5	-	1974.6*	**2221**	×	-	×	×	-	×
`gcd-n-nums`	2.4	7.1	0.4	**9.9**	8.4	11.7	20	1.7	11.7	13.4
`lcm-n-nums`	1.5	17.2	0.4	**19.2**	×	30.2	×	×	30.2	×
`power-root-sqrt`	1.04	93.7	0.3	**95.1**	×	57	×	×	57	×
`primality-sqrt`	438.1	64.9	0.3	503.4	302.9	37.8	**340.7**	×	37.8	×

Table 2. The sketch solving times for three approaches in the presence of a weak model: i) using models (`f-model`) with adherence check, ii) using imperative specification `f-spec`, and iii) synthesis of `main` using synthesized $f_{\texttt{orig}}$. The × values in the table entries denote time-out (> 15 mins).

Benchmark	Solving Time (in s) for Synthesis of `main` using				
	`f-model` + Adherence			`f-spec`	synthesized $f_{\texttt{orig}}$
`bsearch-sort`	8.45	0.87	**9.32**	29.2	83.265
`matrix-exp`	17.14	64.2	**81.34**	×	×
`polyderiv-mult`	5.61	0.9	**6.51**	12.601	×
`polyeval-mult-exp`	2.6	0.9	**3.5**	8.657	10.644

5.3 Comparison with CEGIS and Angelic Synthesis

In this experiment, we compare the performance of our Cegis+ algorithm with that of Cegis and the angelic synthesis algorithm on two metrics: 1) the solving time and 2) the number of synthesis-verification iterations. We expect the Angelic algorithm to perform poorly on benchmarks where the function models are under-constrained and similarly we expect the Cegis algorithm to perform

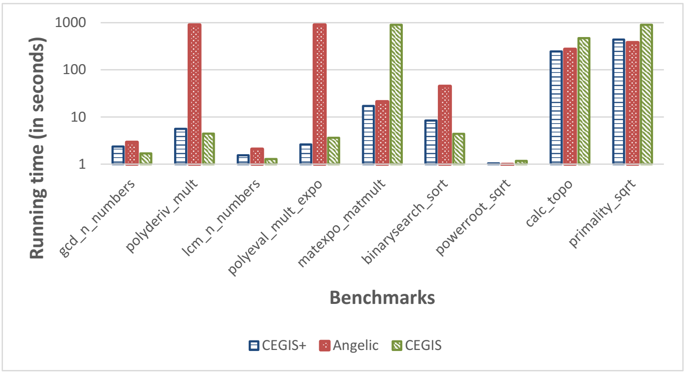


Fig. 6. The solving times of the three algorithms: CEGIS+, Angelic, and CEGIS on the benchmark problems

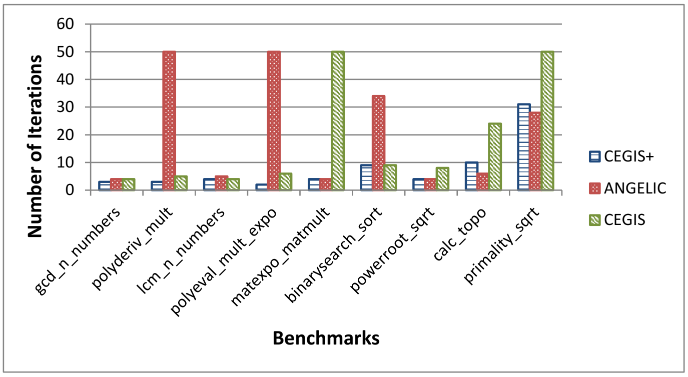


Fig. 7. The number of synthesis-verification iterations performed by the CEGIS+, Angelic, and CEGIS algorithms on the benchmark problems

poorly on benchmarks that are over-constrained. Figure 6 shows the logarithmic graph of running times of the three algorithms on our benchmarks. As expected, we see two benchmarks where the Angelic algorithm times out (set to 15 minutes) whereas the CEGIS algorithm times out on two different benchmarks. The CEGIS+ algorithm solves each of the problem within 440 seconds each and in general has a faster or comparable performance on problems where other algorithms don't timeout. Figure 7 shows the logarithmic graph of the number of synthesis-verification iterations performed by each one of the algorithms on the benchmark problems. The CEGIS+ algorithm performs lesser number of iterations than the CEGIS algorithm for all benchmarks and performs lesser iterations than the Angelic algorithm on all but two benchmarks.

6 Case Study: Boolean DAG Calculator

We present a case study of using function models for synthesizing a calculator that interprets a circuit representing a Boolean DAG (directed acyclic graph). As shown in Figure 8, the interpreter has two main components: a *calculator* and a *parser*, with auxiliary functions like `cmain` that just calls `calc` and `parse`, and `test` that is the test harness.

```
int[n] mtopo(int n, int[2][n] parent)
      models toposort {
  int[n] sorted = topo_uf(n, parent);
  for (int i=0; i<n; i++) {
    int u = sorted[i];
 // node id in sorted must be valid
    assert u>=0 && u<n;
    for (int j=0; j<=i; j++) {
      int v = sorted[j];
 // sorted contains no duplicated node ids
      if (i<j) { assert u != v; }
 // if u occurs after v, u cannot be v's parent
      assert u != parent[v][0] && u != parent[v][1];
    }
  }
}

bit[n] calc(int n, int[2][n] parent, int[n] opr) {
 int[n] sorted = toposort(n, parent);
 bit[n] result;

 for (int i=0; i<n; i++) {
  int u = sorted[i];
  minrepeat { if (opr[u] == ??) {
    result[u] = {| ?? | !result[parent[u][0]] |
      result[parent[u][0]] || result[parent[u][1]]|
      result[parent[u][0]] && result[parent[u][1]]|};
  } }
 }
 return result;
}
```

```
int NOT = 2, OR = 3, AND = 4;

void parse(int n, int[3][n] input,
   ref int[2][n] parent, ref int[n] opr){
// input is an array of 3-tuples
//  of the form {Operator,Src1,Src2}.
// parse converts it to separate parent
//  and opr, and sets unused parent[u][j]
// to -1. parse is also synthesized, using
//  minrepeat, holes, and unknown choices.
}

bit[n] cmain(int n, int[3][n] input){
  int[2][n] parent;
  int[n] opr;
  parse(n, input, parent, opr);
  return calc(n, parent, opr);
}

harness void test(int n, int[3][n] input){
  // test for execution safety
  cmain(n, input);

  // test for functional correctness
  assert cmain(5,{{1},{0},{AND,1,3},
    {OR,0,4},{NOT,1}}) == {1,0,0,1,1};
  // a few more test cases, omitted here
}
```

Fig. 8. The sketch for the Boolean DAG calculator

The `calc` function takes as input a DAG that defines a Boolean circuit, and calculates the value of every Boolean node: the DAG is represented in an *internal representation* consisting of an array `opr` that defines the Boolean operator at each node (we encode the CONST 0 and 1, NOT, OR, and AND operators using integer values), and an array `parent` that denotes the source operands of each node's operator, i.e. `parent[u][j]` stores the node id of node `u`'s `j`-th operand. We assume at most 2 operands for each operator for simplicity and in the case where a node `u` has fewer than 2 operands, some `parent[u][j]` will be set to -1. The output of `calc` is `result`, a bitvector of size `n`, where `result[u]` stores the calculated value of node `u`.

We first need to get a topological order of the DAG to calculate the node values. The simplest imperative `toposort` function (omitted here) is too complex for the solver to reason about, but the declarative model of `toposort` (`mtopo`) is simple and solver-friendly. The main part of `calc` is for calculating each node's value according to the node operator and is based on the previously calculated node values. This calculation is usually performed using a "big switch" (or several `if` statements). The cases for different operators are very similar: depending on the operator, fetch the values of different number (can be 0) of parents, and calculate the result, which are tedious to write. Here `calc` relies on synthesis to reduce this burden (see the `minrepeat` block): it abstracts the common structure of all cases and leaves the differences to unknown constant choices, which are solved by the synthesizer. The use of synthesis also allows the function to adapt to small changes in its requirements. For example, if the programmer decides to no longer support OR because it is redundant with AND and NOT, the synthesizer can adjust the function accordingly without the need to modify the code. Similarly, new operators can be added or encoding of existing operators can be modified just as easily.

The `parse` function takes as input the more readable format of the DAG (where the operator and operands for each node are grouped together as a 3-tuple), and converts it to the internal representation used by `calc`. It needs to copy the right number of operands from `input` to `parent`, and set the remaining `parent` values to -1 depending on the kind of operator, which we specify as choices to be synthesized. The body of `parse` (omitted here) is also sketched with unknown choices to solve similar to `calc`.

An interesting question in this case is how to provide a specification. The `test` function is a harness testing two aspects of the program: execution safety (for any input the program should execute without any assertion failure, array out of bounds error, or reading uninitialized value error) and functional correctness (for a set of known inputs the program should produce the known correct outputs). The two aspects together are sufficient for the Sketch solver to determine all the unknown constants.

As we can see from Table 1 and Figure 6, the use of function models enable the synthesis for this complex program. Without the model, the solver timed out after 5 hours and couldn't synthesize the program; whereas with the model, it solves the program in about 5 minutes (less than 40 minutes even after adding the adherence checking time). We can also see that the CEGIS+ algorithm is much faster than CEGIS because the inputs to the function model `mtopo` are significantly influenced by the unknown holes in `parse`, and CEGIS+ performs slightly better than the pure angelic model.

7 Related Work

The idea of using function models for synthesis is very related to the work on component-based synthesis and is inspired from modular reasoning techniques used in verification. The work on efficiently solving QBF (Quantified Boolean Formulas) is also related to our technique of solving Adherence constraints. We briefly describe some of the related work in each of these areas.

Component-Based Synthesis: The work on component-based synthesis considers the problem of synthesizing larger systems using components as building blocks, which is a central motive for our work of introducing function models in Sketch. The closest related work to ours is that of synthesizing loop free programs using a library of components [11]. This work assumes that all library components have complete logical specifications and it employs a constraint-based synthesis algorithm similar to the angelic algorithm for solving the Correctness constraint in the synthesis phase. Recently this approach has been applied for synthesizing efficient SIMD implementation of performance critical loops [1]. As we have observed for many benchmark problems, often times a partial specification of the library component suffices for synthesizing the correct client code. In the presence of partial function specifications (under-constrained specifications), the angelic algorithm may not converge and is inefficient, whereas the Cegis+ algorithm efficiently converges to the solution for both partially-specified and fully-specified function models. The work on LTL synthesis from libraries of reusable components [15] assumes that the components are specified in the form of transducers (finite state machines with outputs). Our work, on the other hand, considers the problem of functional synthesis and uses constraint-based synthesis algorithms.

Efficient QBF Solving: Efficient solving of Quantified Boolean Formulas (QBF) has been a big research challenge for a long time and the constraints generated by Sketch are too large for current state-of-the-art QBF solver to handle [24]. Recently, word-level simplifications (inspired from automated theorem proving and model finding techniques based on sketches) have been proposed to handle quantified bit-vector formulas in an SMT solver [33]. We can also use this technique to solve the Adherence constraint, but it would require us to provide function templates for the unknown uninterpreted function. Our reduction allows the Cegis algorithm to efficiently solve the constraint without the need of a function template.

Compositional Verification: The idea of using function models for synthesis is inspired from modular verification techniques used for model checking [8]. This idea of modular reasoning using pre-conditions and post-conditions of functions is widely used today in many verification tools such as DAFNY [14] and MAGIC [7] to enable verification of large complex systems. These Assume-guarantee reasoning based techniques applies the divide-and-conquer approach to reduce the problem of analyzing the whole system into verification of individual components [18,30]. For verifying individual components, it uses assumptions to capture the context the component makes about its environment and uses guarantees as properties that will hold after the component execution. It then composes the assumptions and guarantees to prove properties about the whole system. Our function models apply these ideas in the context of software synthesis.

Program Synthesis: Program synthesis has been an intriguing research question from a long time back [16,17]. With the recent advances in SAT/SMT solvers and computational power, the area of program synthesis is gaining a renewed interest. It has been used successfully in various domains such as synthesizing efficient low-level code [26], data-structures [23], string transformations [9,10], table lookup transformations [20] and number transformations [21] from input-output examples, implicit declarative computations in Scala [13], graph algorithms [12], multicore cache coherence protocols [31], automated grading of programming assignments [22], automated inference of synchronization in concurrent programs [32], and for solving games on infinite graphs [2]. We believe our technique can complement the approaches used in many of these domains.

8 Conclusion

In this paper, we presented a technique to perform modular synthesis in SKETCH using function models. This technique enables solving of sketches when they call complex functions and when the correctness of the main harness function depends on a partially interpreted version of the complex function (which we call models). We show that both the CEGIS and the angelic algorithm are inefficient and potentially incomplete, and we present a complete and terminating algorithm to efficiently solve sketches with all kinds of function models. On the basis of promising preliminary results, we believe that this technique will prove very useful in using SKETCH for synthesizing complex and large synthesis problems.

Acknowledgments. We thank Sanjit Seshia and anonymous reviewers for their valuable feedback.

References

1. Barthe, G., Crespo, J.M., Gulwani, S., Kunz, C., Marron, M.: From relational verification to simd loop synthesis. In: PPoPP (2013)
2. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: POPL (2014) (to appear)
3. Bodík, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: POPL (2010)
4. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002)
5. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
6. Celiku, O., von Wright, J.: Implementing angelic nondeterminism. In: Tenth Asia-Pacific Software Engineering Conference (2003)
7. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: ICSE, pp. 385–395 (2003)
8. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16 (1991)
9. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL (2011)

10. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. CACM (2012)
11. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011)
12. Itzhaky, S., Gulwani, S., Immerman, N., Sagiv, M.: A simple inductive synthesis methodology and its applications. In: OOPSLA (2010)
13. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI (2010)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
15. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 395–409. Springer, Heidelberg (2009)
16. Manna, Z., Waldinger, R.: Synthesis: Dreams => program. IEEE Transactions on Software Engineering 5(4), 294–328 (1979)
17. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. 2(1), 90–121 (1980)
18. McMillan, K.L.: A compositional rule for hardware design refinement. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 24–35. Springer, Heidelberg (1997)
19. Seshia, S.A.: Sciduction: combining induction, deduction, and structure for verification and synthesis. In: DAC, pp. 356–365 (2012)
20. Singh, R., Gulwani, S.: Learning semantic string transformations from examples. PVLDB 5 (2012)
21. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 634–651. Springer, Heidelberg (2012)
22. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: PLDI (2013)
23. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: SIGSOFT FSE (2011)
24. Solar-Lezama, A.: Program Synthesis By Sketching. PhD thesis, EECS Dept., UC Berkeley (2008)
25. Solar-Lezama, A.: Program sketching. STTT 15(5-6) (2013)
26. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI (2005)
27. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415 (2006)
28. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: PLDI, pp. 492–503 (2011)
29. Srivastava, S., Gulwani, S., Foster, J.: From program verification to program synthesis. In: POPL (2010)
30. Stark, E.W.: A proof technique for rely/guarantee properties. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 369–391. Springer, Heidelberg (1985)
31. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: Transit: specifying protocols with concolic snippets. In: PLDI, pp. 287–296 (2013)
32. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL. ACM, New York (2010)
33. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods in System Design 42(1), 3–23 (2013)

Synthesis with Identifiers[⋆,⋆⋆]

Rüdiger Ehlers[1,2,3], Sanjit A. Seshia[1], and Hadas Kress-Gazit[2]

[1] University of California at Berkeley, Berkeley, CA, United States
[2] Cornell University, Ithaca, NY, United States
[3] University of Kassel, Germany

Abstract. We consider the synthesis of reactive systems from specifications with identifiers. Identifiers are useful to parametrize the input and output of a reactive system, for example, to state which client requests a grant from an arbiter, or the type of object that a robot is expected to fetch.

Traditional reactive synthesis algorithms only handle a constant bounded range of such identifiers. However, in practice, we might not want to restrict the number of clients of an arbiter or the set of object types handled by a robot a priori. We first present a concise automata-based formalism for specifications with identifiers. The synthesis problem for such specifications is undecidable. We therefore give an algorithm that is always sound, and complete for unrealizable safety specifications. Our algorithm is based on computing a pattern-based abstraction of a synthesis game that captures the realizability problem for the specification. The abstraction does not restrict the possible solutions to finite-state ones and captures the obligations for the system in the synthesis game. We present an experimental evaluation based on a prototype implementation that shows the practical applicability of our algorithm.

1 Introduction

Automatically synthesizing reactive systems from their specifications is an ambitious, yet worthwhile challenge. The applicability of synthesis technology ranges from rapid prototyping to specification debugging, which improves system designer productivity and helps to find incorrect assumptions or forgotten requirements at an early stage in a system development process.

Traditionally, the input and output signals of the systems that are computed in reactive synthesis are purely boolean. If we are not interested in synthesizing hardware, but rather software, this view is often not justified. For example, in a mutual exclusion protocol, we might be getting requests for accesses to a shared resource from a group of clients whose size is unknown a-priori. A robot that satisfies some mission specification on the other hand might need to deliver a large variety of objects. In both cases, we are dealing with *identifier values* that form part of the input or output of a reactive system.

[⋆] This work was partially supported by NSF ExCAPE CCF-1139025/1139138. The first author was also supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no. 259267.

[⋆⋆] A full version of this paper, with appendices and missing proofs, is available at the authors' homepages.

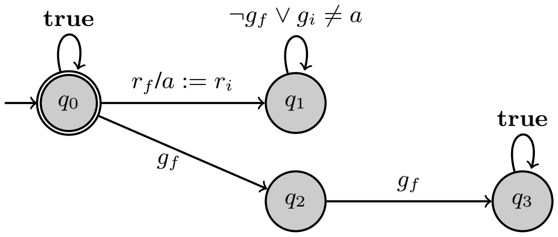


Fig. 1. An example specification (in form of a universal one-weak automaton) for a simple mutual exclusion (mutex) protocol with the input variables r_f (signaling that request is issued) and r_i (representing the identity of the request) and the output variables g_f (signaling that a grant is given) and g_i (for the identity of the grant). The variables r_i and g_i hold values from the domain of identifiers, whereas r_f and g_f are boolean. The automaton models that all requests must eventually be answered by a corresponding grant, and no two grants may be given in successive computation cycles. Accepting states are doubly-circled. As the automaton branches universally, it has to accept along all possible runs for a word to be accepted. Along a transition from q_0 to q_1, we assign a value to variable a that captures that a request with id $a = r_i$ has been issued. In state q_1, a run then waits until the request is answered by a grant, at which point it ends. States q_2 and q_3 ensure that no two grants may be given in successive transitions. All infinite runs for a word must be accepting for the word to be contained in the language of the automaton. Any implementation satisfying this specification is infinite-state, as requests may come in faster than they can be answered. Yet, the specification is realizable as there is no time bound on the answering time.

For the mutual exclusion protocol, the identifiers represent client numbers, whereas for the robot example, they encode the types of the objects that the robot has to deliver.

In this paper, we present an approach to synthesize reactive systems from *specifications with identifier variables*. We present a specification formalism that allows the concise representation of requirements for systems that have identifier input and output variables. By combining universal branching in word automata with identifier variables, we obtain a powerful, yet semantically simple way of describing specifications for such systems. Figure 1 shows an example of such a specification. Automata with universal branching are well-studied in the scope of reactive synthesis as they are a simple, yet expressive, specification model for reactive synthesis algorithms, and at the same time do not blow up under conjunction [10,12]. This is highly desirable as practical specifications describe sets of properties that a system under design all need to fulfill, which are thus connected by conjunction.

The synthesis algorithm that we propose for such specifications exploits the conciseness of our formalism, as it can handle the identifiers in specifications in a symbolic way. The algorithm is sound, but not complete, as the synthesis problem from specifications with identifiers is undecidable. Additionally, our algorithm is always able to detect unrealizable safety specifications.

The core idea of our algorithmic solution is to build a *pattern-based abstraction* of an infinite realizability checking game in which the winning strategies represent correct implementations. The patterns describe constraints over *run points*, i.e., combinations of states in the universal specification automaton that we can be in (along with the corresponding variable valuations) at the same time. In this manner, we reduce dealing with the infinite *concrete realizability game* to solving a finite *abstract realizability game*. In the abstract game, the system player makes promises about how it can control the evolution of a play in the concrete game. In order to compute the possible transitions in the abstract game, we solve a finite-step subgame between an environment player (that sets the next input to the system) and a system player (that sets the next output of the system) in which the system player tries to prove that it can keep its promises. We apply a solver for quantified boolean formulas (QBF) with free variables (ALLQBF) in order to find a compact representation of all moves of the two players in the abstract realizability game that allow the system player to win the finite-step subgame. Solving the abstract realizability game can then be performed using a classical generalized Büchi game solving algorithm [4]. By starting with a small set of patterns, and gradually letting this set grow whenever the abstract game is lost by the system player, we can balance the precision of the abstract game against the computational burden of building it. Often, a small set of patterns suffices, and we exploit this fact in our construction.

The implementations synthesized with our approach are not necessarily finite-state. For example, for the specification given in Figure 1, any implementation satisfying it needs to be infinite-state, and our synthesis algorithm finds one. This distinguishes our approach from classical reactive synthesis methods and classical abstraction-based solution methods for infinite games [7], which can only find finite-state implementations. An implementation that is the outcome of our synthesis approach uses queues as primary datatype to store information about obligations to be fulfilled. As an example, for the specification in Figure 1, the queue would be used to store all requests not having been served yet. While an infinite-state strategy can surely not be exactly implemented with actual hardware, our synthesized implementations use the available memory in a conservative manner, and are thus implementable in practice for input sequences that do not enforce excessive memory usage.

We start by describing our specification modeling framework in Section 2, followed by a theoretical analysis of the corresponding synthesis problem in Section 3. Then, we show how to synthesize from specifications with identifiers with a sound algorithm in Section 4. This algorithm is also guaranteed to detect unrealizable safety specifications. We give some experimental results on a prototype implementation of our approach in Section 5 and conclude with a summary in Section 6.

1.1 Related Work

The benefit of abstracting from concrete data values is well-known in the scope of verification. Wolper [15] defines the notion of *data-independence*, which intuitively means that the control flow of a program only depends on the equalities of the data items handled by the program. He shows how data-independence eases the verification of a large class of properties. His idea is integral to our synthesis approach as all implementations we compute are data-independent.

Previous work on synthesizing systems with infinite input and output domains only considered specification languages that did not permit connecting the values from the infinite domain over time, but rather only allowed local comparisons in every time step. Cheng and Lee [5] present a synthesis approach for cyber-physical systems in which linear-time temporal logic (LTL) as specification logic is extended to allow comparisons of continuous variables such as sensor values as literals. Tabuada [13] considers similar specifications and describes techniques to synthesize systems that not only meet their specification, but do so in a way that is robust to pertubations of the input or output values.

The problem of synthesizing systems with infinite input and output domains is related to solving games with an infinite state space and synthesizing arbitrarily scalable systems. Dimitrova and Finkbeiner [6] discuss the solution of infinite incomplete-information games. They present an abstraction-refinement approach that can find finite winning strategies in such games if these exist. Their approach is not suitable for realizable specifications that have no satisfying finite-state implementation.

Walukiewicz considers the problem of solving parity games over a push-down structure [14]. Winning strategies in such games can be infinite-state, just like in our setting. As push-down games extend the expressible specification by non-regular properties rather than allowing an infinitely-sized input/output alphabet, they are not applicable in the settings dealt with in this paper.

Attie and Emerson describe methods to synthesize arbitrarily scalable systems [1]. Starting with a specification, they propose to synthesize a pair of processes that can then be instantiated as often as needed, and the composition of these processes still satisfies the original specification. As the number of allowed instantiations is not bounded, there is no bound of the state space of the product process. Their composed processes can deadlock in some situations, which is undesirable. Jacobs and Bloem [8] consider the same problem for ring architectures of processes. They show that task to be undecidable for specifications in linear-time temporal logic (LTL), but give a sound semi-algorithm. In contrast to our synthesis algorithm, all of these approaches to synthesize arbitrarily scalable systems cannot deal with specifications that always need an infinite number of states in their implementation regardless of the number of process instantiations, and can also not deal with input/output alphabets with an infinite domain.

2 Modeling Parametrized Specifications

Basics: In this work, we consider *reactive systems with data*, for which the data domains for all input and output signals are either boolean or identifiers. Our reactive system thus has an input signal set $\mathcal{I} = \mathcal{I}_B \uplus \mathcal{I}_I$ that consists of boolean input signals $\mathcal{I}_B$ and signals for reading identifiers $\mathcal{I}_I$, and an output signal set $\mathcal{O} = \mathcal{O}_B \uplus \mathcal{O}_I$ that can be decomposed in the same manner. We call $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ the *interface* of a reactive system. We denote by ID the (infinite) set of identifiers; however, we note that, for the scope of this paper, it is not relevant to fix a concrete domain ID, as we consider equality checks as the only operation on them. In our examples, we always use integers for simplicity. The system runs in discrete time steps, called *computation cycles*, for an indefinite number of steps, which we abstract from by considering *infinite runs of the system*. Such a run is formally given as a word $w = w_0 w_1 w_2 \ldots$, where for every $i \in \mathbb{N}$, we have $w_i \in \mathcal{IS} \times \mathcal{OS}$ for the *input assignment set* $\mathcal{IS} = (\mathcal{I}_I \to \mathsf{ID}) \times (\mathcal{I}_B \to \mathbb{B})$ and the *output assignment set* $\mathcal{OS} = (\mathcal{O}_I \to \mathsf{ID}) \times (\mathcal{O}_B \to \mathbb{B})$. For example, the following word represents a run of a reactive system for the specification in Figure 1:

$$w = \begin{pmatrix} r_f \mapsto \mathbf{false} \\ r_i \mapsto 0 \\ g_f \mapsto \mathbf{false} \\ g_i \mapsto 0 \end{pmatrix} \begin{pmatrix} r_f \mapsto \mathbf{true} \\ r_i \mapsto 5 \\ g_f \mapsto \mathbf{true} \\ g_i \mapsto 5 \end{pmatrix} \begin{pmatrix} r_f \mapsto \mathbf{true} \\ r_i \mapsto 3 \\ g_f \mapsto \mathbf{false} \\ g_i \mapsto 0 \end{pmatrix} \begin{pmatrix} r_f \mapsto \mathbf{false} \\ r_i \mapsto 0 \\ g_f \mapsto \mathbf{true} \\ g_i \mapsto 3 \end{pmatrix} \ldots \tag{1}$$

Formally, we can specify the *behavior* of a reactive system by a function $f : \mathcal{IS}^+ \to \mathcal{OS}$ that maps input histories to an output to produce next. If for a word w, we have that for all $i \in \mathbb{N}$, $f(w_0|_{\mathcal{IS}}\, w_1|_{\mathcal{IS}} \ldots w_i|_{\mathcal{IS}}) = w_i|_{\mathcal{OS}}$, then we say that w is a *run* of f.

Specifications: Given some reactive system interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, a specification is a language $L \subseteq (\mathcal{IS} \times \mathcal{OS})^\omega$. Given some reactive system behavior function $f : \mathcal{IS}^+ \to \mathcal{OS}$, we say that f satisfies L if all runs of f are contained in L. The realizability problem for a language L is to check for the existence of such a behavior function f, and the synthesis problem is to obtain a representation of such a function f (if it exists).

Universal semi-one-weak automata for specifications: To represent specifications over words of infinite length (and an infinite number of identifiers to choose from) in a finitely-representable way, we use *universal semi-one-weak automata* with identifier variables. Formally, for some system interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, such an automaton is given as a tuple $\mathcal{A} = (Q, S, \delta, q_{init}, F)$, where Q is a finite set of states, $S : Q \to 2^{\mathsf{Var}}$ is a scoping function that describes which variables are defined in which states for some domain of identifier variables Var, $q_{init} \in Q$ is the initial state such that $S(q_{init}) = \emptyset$, F is a set of accepting states, and δ is a finite set of transitions.

Every transition is of the form (q, C, A, q'), where $q \in Q$ is a source state, q' is the target state, C is a set of conditions, and A is a set of assignments. A condition is either of the form $v_1 \neq v_2$, $v_1 = v_2$, $b = \mathbf{true}$, or $b = \mathbf{false}$ for some $b \in \mathcal{I}_B \uplus \mathcal{O}_B$ and $v_1, v_2 \in (S(q) \uplus \mathcal{I}_I \uplus \mathcal{O}_I)$. An assignment is of the form (v, u), where $v \in \mathsf{Var}$, $v \in S(q') \setminus S(q)$, and $u \in \mathcal{I}_I \uplus \mathcal{O}_I$; intuitively, u is copied into variable v. For a transition to be valid, we require that $S(q') = S(q) \uplus \{v \in \mathsf{Var} \mid \exists t \in \mathcal{I}_I \uplus \mathcal{O}_I : (v, t) \in A\}$, and every variable may only occur in A once. Along a sequence of transitions in the automaton, every variable may only be assigned once, and the aim of introducing a scoping function into the automaton definition is to make explicit which variables are defined in which states.

The automata that we are concerned with in this paper are *semi-one-weak*, i.e., are like *weak automata* for all accepting states, and are *one-weak* for all non-accepting states, which we also call *rejecting states* in the following. Formally, $\mathcal{A}$ is weak if we can partition Q into a finite number of subsets that are partially ordered by some comparator $\leq_Q$ such that every subset contains only accepting states or only non-accepting states, and for every transition $(q, C, A, q') \in \delta$, for K_q being the partition element that q is in, and $K_{q'}$ being the partition element that q' is in, we have $K_q \leq_Q K_{q'}$. For our semi-one-weak automata, we furthermore require that every rejecting state is one-weak, i.e., it is the only element in its partition. Informally, this means that the only looping paths in the automaton that contain a rejecting state consist solely of self-loops in the rejecting state.

Words $w = w_0 w_1 \ldots \in (\mathcal{IS} \times \mathcal{OS})^\omega$ induce runs in the automaton, where every point in the run is a combination of a state of $\mathcal{A}$ that the run is in, and a variable valuation for the variables in the scope of the state. Formally, every *run point* is thus an element of $\Pi = \{(q, f) \in Q \times (\mathsf{Var} \rightharpoonup \mathsf{ID}) \mid \mathrm{domain}(f) = S(q)\}$, and we say that some sequence $\pi = \pi_0 \pi_1 \ldots \pi_n \in \Pi^*$ is a finite run if $\pi_0 = (q_{init}, \emptyset)$ and for all $i \in \{0, \ldots, n-1\}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta_\Pi$, and that some sequence $\pi = \pi_0 \pi_1 \ldots \in \Pi^\omega$ is an infinite run if $\pi_0 = (q_{init}, \emptyset)$ and for all $i \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta_\Pi$. In both cases, the relation $\delta_\Pi \subseteq \Pi \times (\mathcal{IS} \times \mathcal{OS}) \times \Pi$ describes the possible transitions in a run on a semantic level. It is defined to consist of all tuples $((q, f), x, (q', f'))$ such that there exists some automaton transition (q, C, A, q') such that for all $c \in C$, we have $(f, x) \models c$, and $f' = f \cup \{(v \mapsto x(m)) \mid (v, m) \in A\}$. We say that a run of $\mathcal{A}$ is accepting if for some $q \in F$, there are infinitely many indices i such that $\pi_i = (q, f)$ for some variable assignment f or if the run is finite. We say that $\mathcal{A}$ accepts a word w if all runs for w are accepting.

To simplify the presentation, we also represent semi-one-weak automata with IDs graphically as shown in Figure 1. Accepting states (i.e., those in F) are drawn doubly-circled. Transitions are depicted as arrows, labeled by the conditions and actions. For example, the arrow from state q_0 to q_1 in the figure is formalized as $(q_0, \{r_f = \mathbf{true}\}, \{(a, r_i)\}, q_1)$, whereas the self-loop on state q_1 actually represents two transitions, namely $(q_1, \{g_f = \mathbf{false}\}, \emptyset, q_1)$ and $(q_1, \{g_i \neq a\}, \emptyset, q_1)$.

3 An Analysis of the Synthesis Problem

We start our analysis of the synthesis problem for specifications with identifiers on a theoretical level. After some basic definitions, we define synthesis games and establish *determinacy* of the games. Finally, we derive the undecidability of the synthesis problem for specification with identifiers.

3.1 Basic Definitions

Let $w = w_0 w_1 \ldots \in (\mathcal{IS} \times \mathcal{OS})^\omega$ be a word and $\mathcal{A} = (Q, S, \delta, q_{init}, F)$ be an automaton over $\mathcal{IS} \times \mathcal{OS}$. We can arrange all runs π that correspond to w and $\mathcal{A}$ in a *run tree*. Formally, such a run tree is given as a tuple $\langle T, \tau \rangle$ with a prefix-closed set T and a function $\tau : T \to \Pi$ that maps every tree node to a state and a variable valuation at this state. Figure 2 shows a graphical representation for a run tree for the automaton from Figure 1 and the example word in Equation 1. We obtain $\langle T, \tau \rangle$ from $\mathcal{A}$ by letting T be the smallest subset of Π^* that contains $(q_{init}, \emptyset)$ and such that for every $\pi_0 \pi_1 \ldots \pi_n \in T$, we have that $\pi_0 \pi_1 \ldots \pi_n \pi_{n+1} \in T$ for precisely those $\pi_{n+1} \in \Pi$ with $(\pi_n, w_n, \pi_{n+1}) \in \delta_\Pi$. For all $\pi_0 \pi_1 \ldots \pi_n \in T$, we set $\tau(\pi_0 \pi_1 \ldots \pi_n) = \pi_n$.

We say that a run tree is accepting if for every infinite sequence $\pi = \pi_0 \pi_1 \ldots \in \Pi^\omega$, if for all $i \in \mathbb{N}$, we have $\pi_0 \pi_1 \ldots \pi_i \in T$, then there exist infinitely many $i \in \mathbb{N}$ such that for $\tau(\pi_0 \pi_1 \ldots \pi_i) = (q, f)$, we have $q \in F$. By definition, for a word, there exists an accepting run tree if and only if the word is accepted.

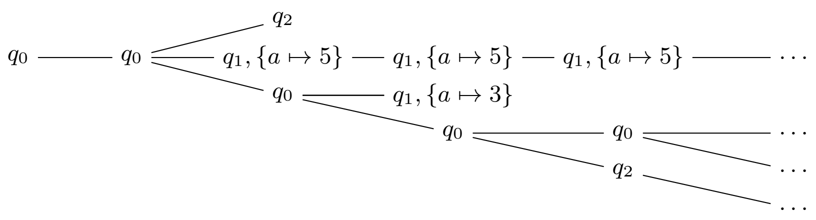


Fig. 2. An example run tree, growing from left to right

3.2 Synthesis Games

A commonly used formalism to study the synthesis problem are *two-player games*. Formally, a game is defined as a tuple $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v^{init}, \mathcal{F})$. We have two players, called player 0 and player 1. Every player p has a set of vertices V^p, a set of actions Σ^p, and an edge function $E^p : V^p \times \Sigma^p \rightarrow V^{(1-p)}$. Without loss of generality, we assume that the initial position v^{init} is an element of V^0. For the scope of this paper, the winning condition $\mathcal{F}$ is defined as a set of subsets of the edges of player 1, i.e., $\mathcal{F} \subseteq 2^{V^1 \times \Sigma^1}$.

In a synthesis game, we declare one player to be the *system player*, whereas the other player is the *environment player*. The system player tries to *win* the game according to the winning condition $\mathcal{F}$, whereas the environment player tries to prevent this.

During the course of the play, the two players alternate in making their moves. They do so by choosing from their respective sets of actions. Afterwards, the position in the game is updated according to the player's edge function, and the play continues. Since the edge functions are required to be total, the play of the game never ends. During the course of the play, the moves of the two players can be collected into their *decision sequence* $\rho = \rho_0^0\rho_0^1\rho_1^0\rho_1^1\rho_2^0 \ldots$, in which for every $i \in \mathbb{N}$ and $p \in \{0,1\}$, we have $\rho_i^p \in \Sigma^p$. The corresponding *play* of the game represents the sequence of positions visited when the two players choose their actions as described in ρ. Formally, a play $\pi = \pi_0^0\pi_0^1\pi_1^0\pi_1^1\pi_0^2 \ldots$ corresponding to ρ is defined as $\pi_0^0 = v^{init}$ and for every $i \in \mathbb{N}$ and $p \in \{0,1\}$, we have $\pi_{i+p}^{1-p} = E_p(\pi_i^p, \rho_i^p)$. We say that a play is winning for player 1 if for all sets of $X \in \mathcal{F}$, player 1 chooses edges in X infinitely often, i.e., there are infinitely many indices $i \in \mathbb{N}$ such that $(\pi_i^1, \rho_i^1) \in X$. Such a winning condition is typically called *transition-based generalized Büchi* for games in which V_0 and V_1 are finite.

Any of the two players in the game can play a *strategy*. Formally, a strategy for player $p \in \{0,1\}$ is a function $f^p : (\Sigma^{1-p})^* \rightarrow \Sigma^p$. We say that a decision sequence $\rho = \rho_0^0\rho_0^1\rho_1^0\rho_1^1\rho_2^0 \ldots$ is in correspondence to some strategy f^p if for all $i \in \mathbb{N}$, we have $\rho_i^p = f^p(\rho_0^{1-p}\rho_1^{1-p} \ldots \rho_{i-1+p}^{1-p})$. If for some strategy f^p, all decision sequences that are in correspondence to f^p induce plays that are winning for player p, then we say that f^p is a winning strategy for player p. We also say that player p *wins the game* whenever it has a winning strategy.

The fact that strategies in games and implementations of systems with identifiers look very similar is no coincidence, as we want to use games to solve synthesis problems – we build games such that the winning strategies for the *system player* in the games *are* in fact the implementations that we are searching for. Starting from a universal semi-one-weak automaton $\mathcal{A} = (Q, S, \delta, q_{init}, F)$, taking player 1 as the system player, and calling player 0 the *environment player*, we build a game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v^{init}, \mathcal{F})$ such that $\Sigma^0 = \mathcal{IS}$ and $\Sigma^1 = \mathcal{OS}$. We furthermore define:

$$\begin{aligned}
V^0 &= 2^{\Pi} \\
V^1 &= V_0 \times \mathcal{IS} \\
E^0(v,x) &= (v,x) \text{ for all } v \in V^0, x \in \Sigma^0 \\
E^1((v,x),y) &= \{(q',f') \in \Pi \mid \exists (q,f) \in v, ((q,f),(x,y),(q',f')) \in \delta_{\Pi}\} \\
&\quad \text{for all } v \in V^0, x \in \Sigma^0, y \in \Sigma^1 \\
v^{init} &= \{(q_0, \emptyset)\} \\
\mathcal{F} &= \bigcup_{(q,f)\in(Q\setminus F)\times(\mathsf{Var}\to\mathsf{ID})} \{\{(X,x,y) \subseteq 2^{\Pi} \times \mathcal{IS} \times \mathcal{OS} \mid \\
&\quad (q,f) \notin X \vee ((q,f),(x,y),(q,f)) \notin \delta_{\Pi}\}\}
\end{aligned}$$

In this game, every position in V^0 intuitively describes a set of run points in the run tree, and E^0 and E^1 ensure that whenever the two players construct some prefix decision sequence $\rho = \rho^0_0\rho^1_0\rho^0_1\rho^1_1 \ldots \rho^0_k\rho^1_k$, then for v being the position reached in the game along a play for this sequence, v is precisely the set of run points that are at level k in the run tree for a word starting with $(\rho^0_0, \rho^1_0)(\rho^0_1, \rho^1_1) \ldots (\rho^0_0, \rho^1_0)(\rho^0_k, \rho^1_k)$. Thus, we can intuitively read off the complete run tree for a decision sequence from its induced play. The winning condition $\mathcal{F}$ then characterizes the set of run trees for which along no branch we eventually get stuck in a run point for a rejecting state. This ensures that precisely the decision sequences that have winning plays in the game are accepted by the specification automaton, and thus the game can be called the *synthesis game* for $\mathcal{A}$. Note that we used the semi-one-weakness of our specification automaton and the fact that variable values never change along a run of the automaton in the definition of the winning condition. Without these facts, the winning condition would need to trace the history of a run point in order for the winning plays in the game to represent the traces that satisfy the specification from which we built the game. The winning condition could not be simply concerned with the edges that are taken infinitely often along a play in the game then.

Lemma 1. *Let $\mathcal{A}$ be a specification automaton over some interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, and $\mathcal{G}$ be a game built from $\mathcal{A}$ and the interface according to the definitions above. If and only if $\mathcal{G}$ is winning for player* 1*, there exists an implementation with interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ all of whose runs are in the language described by $\mathcal{A}$. Furthermore, the winning strategies for player* 1 *in $\mathcal{G}$ are such implementations.*

Determinacy of synthesis games: An important question in game theory is whether a class of games is *determined*, i.e., whether any game in the class admits a winning strategy for one of the players. By the connection between semi-one-weak automata with identifiers and their corresponding games established by Lemma 1, determinacy of all games of the form described above implies that our synthesis problem is actually well-posed: for every specification, there is either an implementation, or we can (theoretically) prove that none exists.

Martin [11] showed that every two-player game for which the winning plays for one of the players form a *Borel set* is determined. This argument is not directly applicable to the type of games built here, as the set of Borel sets is only closed under countable unions/set intersections, but as the identifier domain is infinite, the set of positions in synthesis games can be non-countable. However, note that any identifier value used as input or output of the two players that did not yet occur in the prefix decision sequence in a game always has the same effect on whether a play is going to be winning or not. Thus, we can restrict both players to use fresh identifiers in a certain order (e.g., in increasing order when using integer identifiers) without changing any property of the game, except for the fact that in every position, the two players now only have a finite set of possible moves. This makes the set of positions in the game countable and it can then be shown that the winning plays for any of the players is a Borel set.

Undecidability of synthesis from semi-one-weak automata with identifiers: Despite the simplicity of our specification framework for systems with identifiers, its synthesis problem is unfortunately undecidable. Intuitively, the reason is that we can translate a Turing machine description to a specification that is unrealizable if and only if the Turing machine halts on the empty input tape – the environment in this context provides a sequence of identifiers that serve as addresses on the tape, and the system is required to output the sequence of Turing tape computations along with the Turing machine state. By requiring that an accepting Turing machine state must never be reached, we connect the realizability problem with Turing machine acceptance.

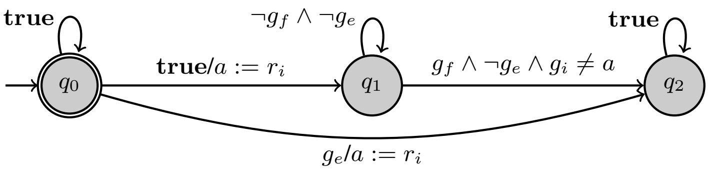
Theorem 1. *Realizability checking for specifications expressed as semi-one-weak universal automata with identifiers is undecidable.*

4 Synthesis Algorithm

As the realizability problem for specifications represented as semi-one-weak universal automata with identifier variables is undecidable, we can only rely on sound, but incomplete, methods to perform synthesis for such specifications. The main idea pursued in the following is to build a finite game that abstracts from details in the synthesis games defined in Section 3.2. The fact that the only data type we consider in this paper are identifiers comes to our rescue at this point, as the single operation that needs to be supported for them is checking for equality. To characterize a situation in the game, it thus suffices to state the equalities of the variable valuations in different run points by which a game position is labeled. We combine this idea with *sound overapproximation* of game situations to ensure the correctness of the computed implementations.

4.1 Patterns

Let $\mathcal{A}$ be a universal automaton with identifiers, and $\mathcal{G}$ be the game built from $\mathcal{A}$ according to the construction from Sect. 3.2. If for a position $v \in V_0$, changing the initial



true
$\neg g_f \wedge \neg g_e$
true
q_0
$\mathbf{true}/a := r_i$
q_1
$g_f \wedge \neg g_e \wedge g_i \neq a$
q_2
$g_e/a := r_i$

Fig. 3. Example specification for an interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ with $\mathcal{I}_B = \emptyset$, $\mathcal{I}_I = \{r_i\}$, $\mathcal{O}_B = \{g_e, g_f\}$, and $\mathcal{O}_I = \{g_i\}$

position to v leads to the game being losing for the system player, then v is called a *bad position*, as once the game reaches v in a play, the system player has no strategy to win. Note that the definition of the games considered here makes sure that if some position v is a bad position, then some other position v' that we can obtain by taking a bijective function $g : \mathsf{ID} \rightarrow \mathsf{ID}$, and replacing every identifier i in v by $g(i)$, is also a bad position, as the concrete values of the identifiers do not matter in our setting, and only their equivalences are of importance. This observation gives rise to the idea of abstracting positions into *pattern*s.

Consider the example specification in Figure 3. In the game that is built according to the construction from Sect. 3.2 from the specification, the position $\{(q_0, \emptyset), (q_1, \{a \mapsto 1\}), (q_1, \{a \mapsto 2\})\}$ is losing for the system player. This can be seen from the fact that from that position, either g_f or g_e have to be set to $\mathbf{true}$ by the system player in order to eventually leave the run points $(q_1, \{a \mapsto 1\})$ and $(q_1, \{a \mapsto 2\})$, with q_1 being rejecting. Since choosing $g_e = \mathbf{true}$ would lead to the transition from q_0 to q_2 being taken, and choosing $g_f = \mathbf{true}$ would lead to taking the transition to q_2 as g_i cannot be 1 and 2 at the same time, we cannot avoid transitioning to q_2, from where we reject a suffix run of the automaton. By the fact that we could replace the concrete identifiers by other values that keep the relationship between the items, and obtain an equally losing position, we call $P = \{(q_0, \emptyset), (q_1, \alpha_1), (q_1, \alpha_2)\}$ a *bad pattern*, as every position that represents an instantiation of this pattern (by substituting the variables α_1 and α_2 by concrete, distinct identifiers values) is losing for the system player.

Note that bad patterns describe sufficient conditions for losing a game. If P is a bad pattern in a game, then the pattern tells us that any position for which we find an instantiation of the bad pattern in its run point set is losing. This way, for example, also the position $\{(q_0, \emptyset), (q_1, a \mapsto 12), (q_1, a \mapsto 42), (q_1, a \mapsto 123)\}$ is losing, as a bad pattern matches a subset of its run points. This stems from the fact that positions are characterized by the run points for runs in a universal automaton, and the more combinations we have, the more properties does the suffix decision sequence have to fulfill in order for the overall decision sequence to be accepted by the automaton.

4.2 Abstract Games

To solve the synthesis problem for a universal semi-one-weak specification automaton $\mathcal{A} = (Q, S, \delta, q_{init}, F)$ with identifiers, we take the *concrete synthesis game $\mathcal{G}$* built from the specification according to Sect. 3.2, and build an *abstract game $\mathcal{G}_A$* from $\mathcal{G}$ that is finite, and thus can be solved by practical game solving algorithms. Every position in the abstract game is labeled by a set of *forbidden patterns*. The abstract position then represents all concrete game positions for which we cannot instantiate any forbidden pattern in the run points by which the concrete game position is labeled. In every abstract position, we require the system player to have suitable next moves for *every* corresponding concrete position. Thus, if the system player can win the game, we know that the specification is realizable.

Patterns can be arbitrarily large, as they can have an arbitrary number of elements. To obtain a finite number of game positions with this idea, we only take patterns from a finite *base set of patterns* $\mathcal{M}$ into consideration. We can, for example, define $\mathcal{M}$ to be the set of all patterns with $\leq b$ elements for some $b \in \mathbb{N}$. With a restricted base set of patterns, our game is only approximate. To achieve the soundness of a synthesis approach based on this idea, we have to ensure that the approximation does not restrict the environment player in any way, and can only put the system player in a disadvantage, as we will do below.

Having only a finite number of positions however does not automatically make the game finite. In fact, the action sets in $\mathcal{G}$ are also infinite. As a remedy, in our abstract game, the two players make *abstract decisions* for their identifier and boolean signals. We use an abstraction that is both simple and powerful: the environment player chooses a subset of the specification automaton transitions as its move, while the system player declares the next forbidden patterns and the states for which it wants to make *progress*.

The idea here is to let the two players announce the *effect* of their choice of moves in the synthesis game rather than giving concrete identifier values, i.e., which automaton transitions are enabled by the move of the environment, and what the successor pattern set is. This idea reduces the two player's decisions to a *finite* domain.

Recall that for a transition $(q, C, A, q') \in \delta$ to fire, *all* constraints in C have to be fulfilled. We call a transition *semi-enabled* (by the environment player) for some choice of boolean and identifier input signal valuations if all constraints over the input signals are satisfied. For an environment player's move to be legal from a position v in the abstract game $\mathcal{G}_A$, there has to exist some position in $\mathcal{G}$ that satisfies all of the constraints of the patterns by which v is marked and some identifier input signal valuation such that the transitions chosen by the player are semi-enabled by the input signals.

After the environment player has made its move, it is the system player's task to (1) choose a set of successor patterns and (2) declare for which rejecting states it wants to perform progress on leaving them. We say that a state is left at a point in the run of the automaton if either the run is not in that state at the point considered, or the state's self-loop is not taken. Consider for example the excerpt from the synthesis game depicted in Figure 4 that we built from the specification in Figure 1. The system player, who owns the left-most position in the figure, can enforce to leave run point $(q_1, \{a \mapsto 3\})$ by choosing $g_f \wedge g_i = 3$ as the next move, and it can enforce to leave run point $(q_1, \{a \mapsto 5\})$ by choosing $g_f \wedge g_i = 5$. Thus, it can declare to be able to make progress on leaving (any run point for) state q_1 and to transition to a position in which the patterns $\{(q_1, \{a \mapsto \alpha_1\}), (q_1, \{a \mapsto \alpha_2\})\}$ and $\{(q_3, \emptyset)\}$ cannot be instantiated.

The system player has to play conservatively, i.e., choose its move while taking into account *every* concrete position that satisfies the constraints imposed by the patterns in v and any of the environment player's concrete input signal values for which the

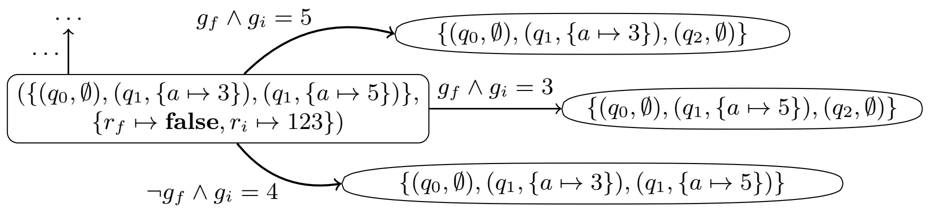

. . .
. . .
$g_f \wedge g_i = 5$
$\{(q_0, \emptyset), (q_1, \{a \mapsto 3\}), (q_2, \emptyset)\}$
$(\{(q_0, \emptyset), (q_1, \{a \mapsto 3\}), (q_1, \{a \mapsto 5\})\},$
$\{r_f \mapsto \textbf{false}, r_i \mapsto 123\})$
$g_f \wedge g_i = 3$
$\{(q_0, \emptyset), (q_1, \{a \mapsto 5\}), (q_2, \emptyset)\}$
$\neg g_f \wedge g_i = 4$
$\{(q_0, \emptyset), (q_1, \{a \mapsto 3\}), (q_1, \{a \mapsto 5\})\}$

Fig. 4. An excerpt from a concrete synthesis game. Positions of player 0 are drawn as ellipses, while the position of player 1 is denoted as a rectangle.

environment player's chosen transition set is valid. In all of these possible cases, there has to be some concrete move of the system player that ensures that the resulting successor position in the concrete game satisfies the patterns declared by the player, and at the same time, progress can be performed by leaving any run point for the states declared.

Let us now formalize $\mathcal{G}_A$ using these ideas. The specification automaton $\mathcal{A}$ is given for some interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, and we have a finite set of base patterns $\mathcal{M}$. A pattern is a set of elements of Θ, where $\Theta = Q \times (\mathsf{Var} \rightharpoonup \{\alpha_i\}_{i \in \mathbb{N}})$ is the set of *pattern atoms*. Without loss of generality, we assume that for every pattern atom (q, f), the domain of f is $S(q)$, and for all $P \in \mathcal{M}$, the set $\{i \in \mathbb{N} \mid \exists (q, f) \in P, e \in S(q) : f(e) = \alpha_i\}$ is of the form $\{0, 1, \ldots, j\}$ for some $j \in \mathbb{N}$. Formally, we define $\mathcal{G}_A = (V_A^0, V_A^1, \Sigma_A^0, \Sigma_A^1, E_A^0, E_A^1, v^{init}, \mathcal{F})$ with the following properties (using the function Post as a placeholder to be explained below):

$$
\begin{aligned}
V_A^0 &= 2^{\mathcal{M}} \cup \{\perp, \top\} \\
V_A^1 &= 2^{\mathcal{M}} \times \Sigma_A^0 \\
\Sigma_A^0 &= 2^{\delta} \\
\Sigma_A^1 &= 2^{\mathcal{M}} \times 2^{Q \setminus F} \\
E_A^0(v, X) &= (v, X) \text{ for all } v \in 2^{\mathcal{M}}, X \in \Sigma_A^0 \\
E_A^1((v, X), (Y_P, Y_D)) &= \mathsf{Post}(v, X, Y_P, Y_D) \text{ for all } (v, X) \in V_A^1, (Y_P, Y_D) \in \Sigma_A^1 \\
\mathcal{F} &= \{V_A^1 \times 2^{\mathcal{M}} \times H_q \mid q \in (Q \setminus F), \\
&\qquad H_q = \{Q' \subseteq Q \setminus F \mid q \in Q'\}\} \\
v^{init} &= \mathcal{M} \setminus \{\emptyset, \{(q_0)\}\}
\end{aligned}
$$

The special positions $\top$ and $\perp$ are declared to be winning/losing for the system player, respectively, so that no successors positions of them need to be defined. All the work in updating the position in the game is deferred to the function Post. Evaluating this function is done in multiple steps. The first step is to check if the environment/input player (player 0) chose a valid move, i.e., if there exists a concrete position that is described by v for which the input player can semi-enable X. Otherwise, the move makes no sense, and we transition to position $\top$, which is a sink (i.e., has no outgoing transitions) and represents that player 0 has made a faulty move.

Then, the *Post* operator checks the system player's move. As the system player declares which patterns should not be instantiable in the next concrete position and along which run points it can promise progress, we move to position $\bot$ whenever the system player is promising too much. In particular, the system player should be able to keep the promise for all position/concrete input combinations for which the environment player's move is valid. In all of these cases, there has to exist some concrete output that leads to a position that does not allow to instantiate any of the promised patterns. At the same time, the system player should be able to make progress with respect to any of the promised run points without violating one of these promised patterns. More formally, the Post function is defined as follows:

– We have $\mathsf{Post}(v, X, Y_P, Y_D) = \top$ **if there does not** exist some $P \subseteq \Pi$ and $x \in \mathcal{I}_I \times \mathcal{I}_B$ such that:
  - no pattern of v can be instantiated in P, **and**
  - X is the set of transitions that are semi-enabled by P and x.
– We define $\mathsf{Post}(v, X, Y_P, Y_D) = \bot$ if for every $P \subseteq \Pi$ and $x \in \mathcal{IS}$ such that
  - no pattern of v can be instantiated in P, **and**
  - X is the set of transitions that are semi-enabled by P and x,

  we do not have that for every run point $\pi_D = (q, f) \in P$ with $q \in Y_D$, there exists some $y \in \mathcal{OS}$ such that:
  - no pattern in Y_P can be instantiated in $\{\pi' \in \Pi \mid \exists \pi \in P : (\pi, (x, y), \pi') \in \delta_\Pi\}$, and
  - we have that $\{\pi' \in \Pi \mid (\pi_D, (x, y), \pi') \in \delta_\Pi\}$ is empty.
– We have $\mathsf{Post}(v, X, Y_P, Y_D) = Y_P$ in all other cases.

Theorem 2. *Let $\mathcal{A}$ be a specification for some interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, and $\mathcal{G}_A$ be the abstract game with initial state v^{init} built from $\mathcal{A}$. If $\mathcal{G}_A$ is winning for player 1 from v^{init}, then there exists an implementation $f : \mathcal{IS}^* \to \mathcal{OS}$ such that all words w that are runs of f are accepted by $\mathcal{A}$.*

Proof. To prove the claim, we show how f can be implemented from a strategy in $\mathcal{G}_A$ that is winning for player 1. We describe f as a program that maintains two data structures: (1) the `set` of run points of $\mathcal{A}$ for the prefix of the decision sequence w observed so far, and (2) a `queue` of run points over non-accepting states in which run points for non-accepting states are queued. The implementation always keeps the set up-to-date and uses the queue for scheduling which run points of non-accepting states are to be left next. By cycling through all of them in the queue, it is ensured that we never get stuck in one of these run points along w, so that w is accepted by $\mathcal{A}$. Additionally, f traces the current position p in $\mathcal{G}_A$.

Let f' be a strategy for player 1 to win $\mathcal{G}_A$ from v^{init}. Our implementation f works as follows: Whenever the implementation obtains a new next input $x \in \mathcal{I}_B \times \mathcal{I}_I$, it computes the set of transitions X that the input x semi-activates from the current position p. Let (Y_P, Y_D) be the move that f' performs for X from p. The implementation then computes a concrete output that leads to leaving the first run point in the queue for a state in Y_D. By the definition of Post, it is made sure that we can always find a concrete output such that additionally, the successor run point set is allowed by Y_P. Note that this computation can be performed in finite time, as we are only concerned with finite sets.

As f' is winning, this means that for all non-accepting states q, we infinitely often have $q \in Y_D$ along the play. Thus, every run point for a non-accepting state is eventually left, and for a semi-one-weak automaton $\mathcal{A}$, this means that w is accepted by $\mathcal{A}$. □

4.3 Computing the Transitions in the Abstract Games

While the abstract games described above only have a finite number of positions and thus can be analyzed by standard algorithms for generalized Büchi game solving [4], we only shifted the problem of dealing with an infinite number of positions in the realizability game to dealing with sets $P \subseteq \Pi$ of run points with an unbounded size in the definition of Post. To effectively compute Post in a practical realizability game building algorithm, we have to reduce reasoning about these sets to efficiently decidable problems. Note that if we manage to only reason about sets P of a bounded size, then this already suffices – as only equality and inequality of identifier values matter, we can then simply enumerate all possible equivalence relations between the identifier values.

So it remains to reduce reasoning about such sets P of unbounded size to reasoning about a bounded number of identifier values. First of all, consider checking if $\mathsf{Post}(v, X, Y_P, Y_D) = \top$ holds. We can restrict our search for P to sets of cardinality $|X|$, as we only need at most one run point per transition in order to semi-enable it, and having more run points only makes it harder to ensure that no pattern in v can be instantiated in P. In fact, we can even restrict our search to having precisely one run point (q, f) for every transition in X such that the transition starts from q.

Testing if $\mathsf{Post}(v, X, Y_P, Y_D) = \bot$ holds while only quantifying over finite sets is a bit more difficult. We apply the following idea in order to avoid having to reason over very large sets P in order not to sacrifice soundness. We again consider sets P of cardinality $|X|$ as above, and require that for every concrete input $x \in \mathcal{IS}$ such that X is the set of transitions activated from P and for every run point (q, f) in P with $q \in Y_D$, there exists a concrete output $y \in \mathcal{Y}$ such that the run point (q, f) is left under (x, y). Additionally, we quantify over all run point sets P' of size b_{max}, where b_{max} is the largest size of a pattern in $\mathcal{M}$, and require that from the concrete position P', we do not reach a position through (x, y) in which some pattern in Y_P can be instantiated if $P \cup P'$ does not violate a pattern in v and if not more than $|X|$ transitions are semi-enabled from $P \cup P'$ for X. The idea here is that the system player has to come up with a move that allows leaving any possible run points for the non-accepting states declared in Y_D and that is robust with respect to adding more run points. In a sense, we hide certain run points from the system player, but the system player knows already the patterns that cannot be instantiated in the current concrete position. This allows us to quantify only over sets of size b_{max} in P'. If the system player can choose y such that adding more “surprise” run points does not let it exceed Y_P after the next transition, then the system player has shown that it can make a robust next choice to hold the progress promise Y_D and the successor position promise Y_P after the current round. We only need to consider at most b_{max} predecessor run points for this check as for no pattern, we need more than b_{max} run points before a transition in order to violate it after a transition. By letting player 1 fix its choices before the “surprise” run points are chosen, we only have to quantify over elements in P' once for any possible pattern in Y_P that can potentially be instantiable in the concrete game position after the transition.

As a summary, we use the following *finite-step* game for testing if $\mathsf{Post}'(v, X, Y_P, Y_D) = \bot$ holds, where Post' denotes the approximate version of Post implementing the ideas from above:

1. First, the environment player chooses some run points P (one for each element in X) and concrete input x such that x semi-enables the transitions in X.
2. Then, the environment player chooses some run point $(q, f) \in P$ with $q \in Y_D$ along which the system has to make progress (only if $Y_D \neq \emptyset$).
3. It is then the system player's turn to choose some concrete output y that leads to leaving the run point chosen by the other player (if any). The environment player wins if this is not possible.
4. Finally, the environment player picks b_{max} additional run points P'. If any pattern of Y_P can be instantiated in $\{\pi' \in \Pi \mid \exists \pi \in P \cup P' : (\pi, (x, y), \pi') \in \delta_\Pi\}$ while no pattern in v is instantiable in $P \cup P'$ and X semi-enables the transitions in X for x from $P \cup P'$, the environment player wins. Otherwise the system player wins.

4.4 Applying an (ALL)QBF Solver for Efficient Reasoning in Practice

After we have reduced computing the Post' function (i.e., our approximate version of Post) to a problem over finitely many elements in the previous subsection, it makes sense to discuss how to compute Post' in practice. Observe that testing iff $\mathsf{Post}'(v, X, Y_P, Y_D) = \bot$ holds for some values of v, X, Y_P, and Y_D is the most difficult step and can be formulated as the finite-step game given above. This fact suggests that using a solver for quantified boolean formulas (QBF) is reasonable. We can encode the boolean input and output variables in x and y as simple boolean values. For the identifiers involved in the finite-step game, let $\mathcal{C} = \{c_0, c_1, \ldots, c_n\}$ be the set of identifier variables in the run points and input and output signals involved, and $c_0, c_1, \ldots, c_n$ be the order in which they are introduced. We reserve a family of boolean variables $\{e_{ij}\}_{c_i, c_j \in \mathcal{C}}$ as an *equality matrix* between them that represents which identifier variables point to the same identifier. As equality is an equivalence relation, the matrix $\{e_{ij}\}_{c_i, c_j \in \mathcal{C}}$ must represent such a relation. We assign the task to keep the matrix representing an equivalence relation to the two players in the finite-step game; whenever a player introduces a new variable c_k for $0 \leq k \leq n$ in the game, the player must assign values to $\{e_{ik}, e_{ki} \mid 0 \leq i \leq k\}$ such that $\{e_{ij}\}_{i,j \in \{0,\ldots,k\}}$ is still an equivalence relation.

We encode the QBF instance from the point of view of the system player that asks if for some given v and X, there exists some choice for Y_P and Y_D such that the system player wins the finite-step game explained above. This has the advantage that we can model Y_P and Y_D using *free* variables and apply an ALLQBF [2] solver to compute a boolean formula g that represents all valuations of the free variables that make the quantified boolean formula satisfied. From g, we can then easily enumerate all *Pareto-optimal* moves using a satisfiability (SAT) solver. We call a valuation of Y_P and Y_D Pareto-optimal if no element can be added to Y_D and no element can be added to Y_P such that the resulting valuation of the free variables in the QBF instance is still a model of it. Note that for building the abstract realizability game $\mathcal{G}_A$, we only have to consider the Pareto-optimal choices of the system player as playing non-optimal moves does not help the system player in any way.

For computing the possible values for X that do not let the environment player lose the finite-step game from some position v (i.e., computing whether $\mathsf{Post}'(v, X, Y_P, Y_D) = \top$ holds for arbitrary Y_P and Y_D), we can apply the same equality matrix encoding. However, this time, we only need a SAT solver as there is no quantifier alternation. Again, we only enumerate the Pareto-optimal choices, i.e., the largest elements X that avoid having $\mathsf{Post}'(v, X, Y_P, Y_D) = \top$.

4.5 Completeness for Unrealizable Safety Specifications

Assume that the specification we are concerned with is of safety type, i.e., all rejecting states in the specification automaton have an unconstrained self-loop. As the (concrete) synthesis game is determined, this means that for every unrealizable such specification, there is some number k such that the environment has already won after k steps, and there is only a finite set of positions that might be visited before that. If we add enough patterns to distinguish all of these positions from all respective other positions, then after analyzing the abstract game, we can see that all positions visited before losing the game (i.e., before entering some rejecting state) only characterize one concrete position each. From this fact we can infer that the specification under concern is in fact unrealizable. Thus, by adding a post-solution abstract game analysis step to check if all abstract positions only represent one concrete position each, the algorithm can always detect unrealizable safety specifications.

5 Experimental Results

We implemented our synthesis approach, without the extensions of Sect. 4.5, in a prototype implementation written in Python, using the SAT solver PICOSAT v.957 [3] and the ALLQBF solver GHOSTQ 0.85 [9] as solving engines. As there is no other synthesis tool for infinite-state systems to compare against, in our evaluation, we focus on showing the applicability of our techniques on an example of practical relevance.

Case study: We synthesize a controller to let a robot automatically deliver menu items in a restaurant to guests who ordered them. We partition the floor of the restaurant into a set of regions Z and define the neighborhood relation of regions in the restaurant by an adjacency relation $R \subseteq Z \times Z$. The robot can move between adjacent regions in every computation cycle, and pick up or deliver a food item. Food is always picked up from the same region z_{pickup} (i.e., the kitchen), where we assume a tray with prepared food that is continuously replenished to be located. The first customer orders specific food items, while the other one just requests *any* food item to be delivered. Figure 5 depicts the setting.

For our reactive system, we have as interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ with $\mathcal{I}_B = \{r_{order1}, r_{order2}\}$, $\mathcal{I}_I = \{f\}$, $\mathcal{O}_B = \{m_z \mid z \in Z\} \cup \{deliver\}$, and $\mathcal{O}_I = \{pickup\}$. The specification has the following constraints:

- At every point in time, the robot is only in one region m_z, and if the region changes from one cycle to the next one, the predecessor and successor regions are connected by R.
- Whenever food item i is ordered by customer 1 (i.e., we have $f = i$ and $r_{order1} = \mathbf{true}$), then eventually, the robot picks up food item i from z_{pickup} and does not *deliver* it until it is in region 4 (i.e., $m_4 = \mathbf{true}$) at which point it should *deliver* it.

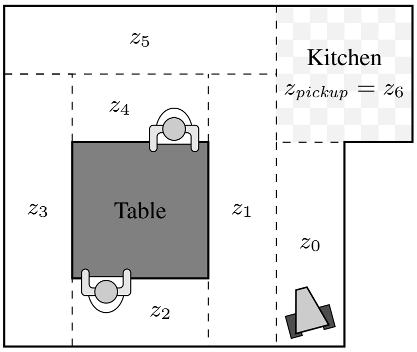


Fig. 5. A restaurant scenario with seven zones and two clients. The robot starts in zone z_0.

- The robot always picks up a food item of kind *pickup* when entering the kitchen.
- The robot must deliver a food item before entering the kitchen again. Deliveries may only take place in regions with customers.
- Whenever customer 2 orders food, then a food item is eventually brought to region 2.
- The robot does not deliver a food item to customer 1 that has not been ordered.
- New orders by a customer are ignored if there are orders by the same customer that have not yet been fulfilled.

We consider one variant of the scenario with the second customer being present, and one variant without that customer. In both cases, around 70 transitions are needed to model the respective scenario.

In addition to the robot scenario, we also considered the mutex protocol from Figure 1 and the example specification from Figure 3.

Results: Table 1 shows the experimental results for building the abstract games. The computation times were obtained on an Intel i5-3230M 2.60GHz computer running an x64-version of Linux. The actual game solving process of the abstract games always took less than 0.1 seconds.

As pattern sets, we always start with all patterns of size at most 1. For the robot waiter scenario, we find the resulting abstract games to be losing for the system player. An analysis of the scenario reveals that the reason is that we have a state q_c^1 that disallows the robot to deliver a menu item to customer 1 that is different to the one previously stored. This state has the task to check that only ordered menu items are delivered. It is entered whenever a menu item is ordered while no request is yet unfulfilled. When entering the state, the menu item requested is stored into the (single) variable in its scope. If we are in this state with two different run points, then there is no menu item that the robot can deliver. However, this is a situation that cannot occur during a play in the concrete synthesis game. By adding the pattern $\{(q_c^1, \alpha_0), (q_c^1, \alpha_1)\}$, this is taken into account in the abstract game and the setting becomes realizable. This modified pattern set is denoted as "1^+" in Table 1.

Table 1. Result table for the prototype implementation of our synthesis approach

Benchmark:	1-client robot waiter		2-client robot waiter		Mutex	Example from Figure 3		
# States:	17		19		4	3		
# Transitions:	68		72		7	6		
Max. pattern size considered:	1	1^+	1	1^+	1	1	2	3
Time to build abstract game:	19m 10.2s	25m 35.5s	19m 11.5s	28m 50.0s	1.08s	0.6s	1.3s	4.5s
Number of positions in abstract game:	174	216	209	255	10	6	4	4
Number of edges in abstract game:	658	792	966	1152	14	7	4	4
Realizable:	✗	✓	✗	✓	✓	✗	✓	✓

For the mutex protocol, taking all patterns of size at most 1 suffices. On the other hand, the example specification from Figure 3 is not found to be realizable with the patterns of size at most one. Thus, we also considered all patterns of size up to 2 (and additionally 3), where we removed patterns that are equivalent to other patterns in the set (such as, e.g., $\{(q_1, \alpha_1), (q_2, \alpha_2)\}$ when $\{(q_1, \alpha_2), (q_2, \alpha_1)\}$ is also present). Starting with a maximum pattern size of 2, the specification is found to be realizable.

It can be seen that the robot waiter scenario can be tackled by our approach, despite the large number of transitions in its specification automaton. While at first, this may seem surprising (after all, the number of different moves for the environment player in the abstract game is exponential in the number of automaton transitions), this success can be attributed to the idea to only enumerate the Pareto-optimal moves and use SAT and ALLQBF solvers as efficient reasoning engines, which reduces the size of the abstract game and the computation times.

6 Conclusion and Outlook

In this paper, we presented the first synthesis approach for specifications with identifier variables that is capable of deriving infinite-state implementations for cases in which these are actually needed. For showing the practical feasibility of our approach, we applied it to a robot waiter scenario. Our work can be seen as one of the first steps towards solving the problem of *reactive synthesis with data constraints*. We focused on identifiers as data type here, as these are relatively simple to handle, and thus suitable for one of the first examinations of the reactive synthesis problem with data. We conjecture that our modeling framework, i.e., universal semi-one-weak automata, remains useful when extending the data domain, as the model is both simple and powerful.

Our prototype implementation uses off-the-shelf SAT and (ALL)QBF solvers and employs a simple equivalence-matrix-based approach to deal with the identifiers in this context. We conjecture that there is still a lot of room for improvement, e.g., by optimizing the QBF encoding and using a special (ALL)QBF solver that is tuned towards finding only the Pareto-optimal variable valuations. Also, a counter-example guided abstraction refinement approach to pattern selection might be suitable.

This work was driven by investigating the class of specifications that can be supported in a practical synthesis algorithm working over an infinite data domain. Thus, the specification class and the solution algorithm are carefully aligned. It would be interesting to examine how the specification class can be further extended (such as by loosening the semi-one-weakness requirement). Additionally, it would be useful to develop a suitable specification logic from which the universal automata can be efficiently generated.

References

1. Attie, P.C., Emerson, E.A.: Synthesis of concurrent systems with many similar processes. ACM Trans. Program. Lang. Syst. 20(1), 51–115 (1998)
2. Becker, B., Ehlers, R., Lewis, M., Marin, P.: ALLQBF solving by computational learning. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 370–384. Springer, Heidelberg (2012)
3. Biere, A.: Picosat essentials. JSAT 4(2-4), 75–97 (2008)
4. Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 153–167. Springer, Heidelberg (2007)
5. Cheng, C.H., Lee, E.A.: Numerical LTL synthesis for cyber-physical systems. CoRR abs/1307.3722 (2013)
6. Dimitrova, R., Finkbeiner, B.: Abstraction refinement for games with incomplete information. In: FSTTCS, pp. 175–186 (2008)
7. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003)
8. Jacobs, S., Bloem, R.: Parameterized synthesis. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 362–376. Springer, Heidelberg (2012)
9. Klieber, W., Janota, M., Marques-Silva, J., Clarke, E.: Solving QBF with free variables. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 415–431. Springer, Heidelberg (2013)
10. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
11. Martin, D.A.: A purely inductive proof of Borel determinacy. In: Recursion theory, Symposium on Pure Mathematics, pp. 303–308 (1982)
12. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
13. Tabuada, P.: Verification and Control of Hybrid Systems. Springer (2009)
14. Walukiewicz, I.: Pushdown processes: Games and model-checking. Inf. Comput. 164(2), 234–263 (2001)
15. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: POPL, pp. 184–193. ACM Press (1986)

Synthesis for Polynomial Lasso Programs

Jan Leike[1] and Ashish Tiwari[2,⋆]

[1] University of Freiburg, Germany
leike@informatik.uni-freiburg.de
[2] SRI International, Menlo Park, CA
ashish.tiwari@sri.com

Abstract. We present a method for the synthesis of polynomial lasso programs. These programs consist of a program stem, a set of transitions, and an exit condition, all in the form of algebraic assertions (conjunctions of polynomial equalities). Central to this approach is the discovery of non-linear (algebraic) loop invariants. We extend Sankaranarayanan, Sipma, and Manna's template-based approach and prove a completeness criterion. We perform program synthesis by generating a constraint whose solution is a synthesized program together with a loop invariant that proves the program's correctness. This constraint is non-linear and is passed to an SMT solver. Moreover, we can enforce the termination of the synthesized program with the support of test cases.

1 Introduction

There have been significant advances in automating program verification, and even extending the verification techniques to perform automated synthesis of correct programs. Often, automation is achieved using appropriate abstract domains for analysis. The choice of abstract domains is governed by the class of program fragments being analyzed. In this paper, we are interested in programs that perform some numerical computation. For reasoning about such programs, the theory of polynomial ideals has proven to be an excellent abstract domain because of two reasons. First, there is a nice correspondence between subsets of the program state space and polynomial ideals (as established in the field of algebraic geometry), and second, there are effective algorithms for computing with polynomial ideals. In this paper, we will use the abstract domain of polynomial ideals for reasoning about polynomial lasso programs.

In our terminology, a polynomial lasso program consists of an assertion describing program states before loop entry, an assertion describing program states after loop termination and a set of transitions corresponding to the branches in the loop body. All involved assertions are algebraic; that is, conjunctions of polynomial equalities.

⋆ Supported in part by the National Science Foundation grant SHF:CSR-1017483 and by DARPA under contract FA8750-12-C-0284. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Our approach for analysis of such polynomial lasso programs is not based on iterative fixpoint computation. Instead, we use the constraint-based approach, also known as template-based approach, for directly finding fixpoints using constraint solving. This way we avoid convergence issues of iterative fixpoint methods. Our starting point is a method presented by Sankaranarayanan, Sipma and Manna [1]. Despite its obvious incompleteness, the method is often successful in verifying programs. Why is this method "complete in practice"? We answer the question here by presenting a first completeness criterion for this method. For this purpose, we have to extend the original invariance criteria in [1] and generate a new and refined *invariance condition*.

Our interest here is not just on the verification problem, but also on the synthesis problem. Specifically, taking inspiration from recent work on synthesis of programs by completing partial program "sketches" [2,3], we start with a polynomial lasso program that contains parameters (variables to be synthesized) and a post condition. The goal is to find values for the parameters that result in a correct program. We solve the synthesis problem by generating a *synthesis constraint*—a constraint whose solution provides a valuation for the parameters. Additionally, the constraint's solution also supplies values that define an inductive loop invariant for the synthesized polynomial lasso program. This invariant constitutes as proof that the synthesized program is in fact correct with respect to the given post condition. Thus, we simultaneously synthesize the program and its proof of correctness. There is one caveat though: if variables that are critical to termination have parameterized updates, then the synthesized lasso program might not be terminating. To solve this problem, we use a finite number of test cases that specify input variable assignment, output variable assignment and a sequence of loop transitions. These test cases are used to strengthen the synthesis constraint so that the undesirable solutions are eliminated.

The template-based approach reduces the synthesis problem and the loop invariant discovery problem into an $\exists\forall$ constraint: the template variables and the synthesis variables are existentially ($\exists$) quantified, whereas the program variables are universally ($\forall$) quantified [3]. We use the theory of polynomial ideals to (conservatively) eliminate the inner $\forall$ quantifier. The resulting formula is our synthesis constraint – an (existentially quantified) conjunction of non-linear algebraic equalities – which is solved by an off-the-shelf non-linear SMT solver.

We demonstrate that the template-based approach on polynomial ideals abstract domain can be used to successfully synthesize polynomial lasso programs. However, the approach has certain limitations. First, it cannot handle inequalities. Polynomial ideals logically correspond to conjunctions of polynomial equalities. Now, inequalities can be encoded as equalities, but algorithms on polynomial ideals (that compute canonical Gröbner basis) do not lift easily to reasoning about the encoded inequalities [4]. For handling inequalities, one could use semialgebraic sets as the abstract domain, and then use algorithms based on either *cylindric algebraic decomposition* [5] or the *Positivstellensatz* [6,7,4], but we leave that for future work.

A second issue is the size of the synthesis constraint. Non-linear solvers scale very poorly with increasing number of variables and the synthesis constraint (generated by the synthesis process) can be large and tends to be non-linear.

The final issue is related to the completeness of our approach. Incompleteness arises due to the use of templates, and also due to the use of polynomial ideal theory rather than the theory of reals. We address the latter issue in section 5. For the former issue, we just have to use polynomial templates with sufficiently large degree bounds. In our examples, a general template of degree two or three was sufficient, but the size of generic template polynomials grows exponentially with their degree.

2 Related Work

The automatic discovery of polynomial invariants for imperative programs has received a lot of attention in recent years. Müller-Olm and Seidl generate invariant polynomial equalities of bounded degree by backwards propagation [8]. This can be seen as an extension to Karr's algorithm [9], which uses only linear arithmetic. Seidl, Flexeder and Petter apply the backwards-propagation method to programs over machine integers, i.e., programs whose variables range over the domain $\mathbb{Z}_{2^w}$ [10].

Rodrígues-Carbonell and Kapur use an iterative approach based on forward propagation and fixed point computation on Gröbner bases over the lattice of ideals to generate the ideal of all loop invariants [11,12].

Colón combines the two aforementioned approaches by doing the fixed point computation on ideals with linear algebra [13]. He introduces the notion of pseudo-ideals to ensure termination of the fixed point computation while retaining the expressiveness of generated invariants.

Polynomial program invariants can also be derived without using Gröbner basis computations [14]. Cachera et al. use backwards analysis and variable substitution on template polynomials for an incomplete approach.

The constraint solving approach that generates invariant polynomial equalities using templates was proposed by Sankaranarayanan, Sipma and Manna [1]. Invariant generation is a central ingredient to our synthesis method, so we want the invariant generation process to be as complete as possible. Therefore we extend their approach by using a more general condition for the invariant (see also Remark 2) that enables us to state a completeness criterion.

Polynomial lasso programs have also received some attention regarding the analysis of their termination properties. Bradley, Manna and Sipma use finite difference arithmetic to compute lexicographic polynomial ranking functions for polynomial lasso programs [15].

All the aforementioned papers consider the verification (or the invariant generation) problem. In this paper, inspired by recent work on program synthesis [2], we also consider the synthesis problem. Our work can be considered a more formal approach to Colón's method [16] that uses non-linear constraint solving to instantiate program schemata (parameterized programs augmented with constraints). Our approach relies on algebraic methods instead of heuristics.

Finally, Srivastava et al. [17] describe a big-picture program synthesis algorithm from scaffolds. These scaffolds consist of pre- and postconditions, a program flow template, and bounds on the number of variables and the number of local branches. For the synthesis condition, all control flows of the template program are unfolded and constraints are generated with respect to invariants and ranking functions ensuring the program's correctness and termination. This constraint is then proven by a specialized external method and our algorithm can be used as one of these external methods.

3 Preliminaries

Let V be a set of variables, $V = \{x_1, \ldots, x_n\}$. The variables of the 'next state' are denoted by the corresponding primed variables $V' = \{x'_1, \ldots, x'_n\}$. Having both primed and unprimed variables in an expression enables stating a relationship between two states.

For the set of real numbers $\mathbb{R}$, let $\mathbb{R}[V]$ denote the ring of polynomials in the variables V with coefficients from $\mathbb{R}$. A subset $I \subseteq \mathbb{R}[V]$ is an *ideal* if (a) $0 \in I$, (b) $f + g \in I$ for all $f, g \in I$, and (c) $h \cdot f \in I$ for all $f \in I$ and $h \in \mathbb{R}[V]$. For a set of polynomials $P = \{p_1(V), \ldots, p_k(V)\}$, *the ideal* $\langle P \rangle$ *generated by* P is

$$\langle P \rangle = \langle p_1, \ldots, p_k \rangle = \left\{ \sum_{i=1}^{k} q_i(V) p_i(V) \,\middle|\, q_1, \ldots, q_k \in \mathbb{R}[V] \right\}.$$

Note that if all polynomials in P evaluate to 0 at any point in $\mathbb{R}^n$, then all polynomials in $\langle P \rangle$ will also evaluate to 0 at that point.

By the Hilbert Basis Theorem, every ideal I has a finite set of generators. Moreover, for a fixed ordering on the monomials (such as total degree lexicographic ordering induced by any precedence relation on the variables), there is a finite "canonical" set of generators of I called a *Gröbner basis.* A Gröbner basis $G = \{g_1, \ldots, g_k\}$ for I has the following properties [18].

1. G is computable in DOUBLE-EXPSPACE from a set of generators of I (Buchberger's Algorithm).
2. For all $p \in \mathbb{R}[V]$, the result of division of p on G, denoted $\mathrm{NF}_G(p)$, is unique and does not depend on the order in which the division steps are performed.
3. For all $p \in \mathbb{R}[V]$, $\mathrm{NF}_G(p) = 0$ iff $p \in I$.

For example, if $P = \{xy - 2, x^2 - 4\}$ and we use the precedence $x \succ y$, then $G = \{x - 2y, y^2 - 1\}$ is a Gröbner basis for the ideal $\langle P \rangle$. Division of p on G can be performed by replacing x by $2y$ and replacing y^2 by 1 in p repeatedly. The result $\mathrm{NF}_G(x^2 + y^2 - 5)$ of division of $x^2 + y^2 - 5$ on G is 0, and hence we can conclude that $x^2 + y^2 - 5 \in \langle P \rangle$.

Definition 1 (Radical Ideal). *An ideal I is a* radical ideal *if $f^m \in I$ implies $f \in I$ for every $m \in \mathbb{N}$.*

Given an ideal I, note that the set $\{f \mid \exists m \in \mathbb{N} : f^m \in I\}$ is a (radical) ideal.

Definition 2 (Algebraic Assertion). *An* algebraic assertion $\varphi(V)$ *(or just* φ*) over the set of variables* V *is a formula of the form* $\bigwedge_{i=1}^{m} p_i(V) = 0$ *where each* $p_i \in \mathbb{R}[V]$ *for* $1 \leq i \leq m$*.*

An algebraic assertion $\bigwedge_{i=1}^{m} p_i(V) = 0$ generates an ideal $\langle\varphi\rangle = \langle p_1, \dots, p_m\rangle$. We will use φ to denote the formula as well as the set of polynomials $\{p_1, \dots, p_m\}$ in the formula. An assertion φ can be interpreted in the theory $\mathbb{R}$ of reals or in the theory $\mathbb{C}$ of complex numbers. A *valuation* is a mapping from variables to values (in the set of real numbers or the set of complex numbers). A polynomial in $\mathbb{R}[V]$ evaluates to a value (in $\mathbb{R}$ or $\mathbb{C}$) for a given valuation for V.

Theorem 1 (Zero Polynomial Theorem). *A polynomial* $p \in \mathbb{R}[V]$ *is zero for all possible valuations* $\nu : V \to \mathbb{R}$ *if and only if all of its coefficients are zero.*

Lemma 1. *Let* φ *be an algebraic assertion over* V *and* $p \in \mathbb{R}[V]$ *a polynomial. If* $p \in \langle\varphi\rangle$*, then* $\mathbb{R} \models \varphi(V) \to p(V) = 0$*.*

Theorem 2 (Hilbert's Nullstellensatz [18]). *Let* φ *be an algebraic assertion and* $p \in \mathbb{C}[V]$ *a polynomial. If* $\langle\varphi\rangle$ *is a radical ideal and* $\mathbb{C} \models \varphi(V) \to p(V) = 0$*, then* $p \in \langle\varphi\rangle$*.*

Lemma 2. *Let* $p, s \in \mathbb{R}[V]$ *and* $\langle\varphi\rangle \subseteq \mathbb{R}[V]$ *be an ideal. Then,* $p \in \langle s, \varphi\rangle$ *if and only if there is a polynomial* $t \in \mathbb{R}[V]$ *such that* $p - t \cdot s \in \langle\varphi\rangle$*.*

Proof. Let $\langle\varphi\rangle = \langle p_1, \dots, p_k\rangle$. By definition, $p \in \langle s, \varphi\rangle$ iff there are $t, t_1, \dots, t_k \in \mathbb{R}[V]$ such that $p = ts + \sum_i t_i p_i$. This is equivalent to $p - ts = \sum_i t_i p_i$, which holds iff $p - ts \in \langle\varphi\rangle$. □

Similar to the definition by Sankaranarayanan et al., we introduce template polynomials as a means for finding polynomials with certain properties. In our definition the template coefficients can be non-linear polynomials. For the mathematical details regarding template polynomials, see [1].

Definition 3 (Template Polynomial). *Let* A *and* V *be two disjoint sets of variables. A* template polynomial *or* template *over* (A, V) *is a polynomial with variables* V *and coefficients from* $\mathbb{R}[A]$*. A template is said to be a* linear template *if all of its coefficient polynomials are linear.*

Template polynomials will be denoted by upper case Greek letters. Given a degree bound d, the *generic template polynomial* Ψ over (A, V) of total degree d is given by

$$\Psi(V) = \sum_{|\gamma| \leq d} a_\gamma V^\gamma$$

where $\gamma \in \mathbb{N}^{\#V}$ is a multi-index and $A = \{a_\gamma \mid \gamma \in \mathbb{N}^{\#V}\}$ are template variables.

Definition 4 (Semantics of Templates). *For a set of template variables* A*, an* A-valuation *is a map* $\alpha : A \to \mathbb{R}$*. This map can be naturally extended to a map* $\tilde{\alpha} : \mathbb{R}[A][V] \to \mathbb{R}[V]$ *that replaces every occurrence of an* $a \in A$ *by* $\alpha(a)$*.*

4 Polynomial Lasso Programs

We define the syntax and semantics of polynomial lasso programs. We also define inductive invariants for such programs. Henceforth, semantic entailment, $\models$, should always be interpreted as in the theory $\mathbb{R}$ of reals.

Definition 5 (Polynomial Lasso Program). *A* polynomial lasso program $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ *consists of*

- *a set of variables V,*
- *an algebraic assertion* stem *over V called the* program stem*,*
- *a* set of transitions $\mathcal{T}$*, where each transition $\tau \in \mathcal{T}$ is an algebraic assertion over $V \cup V'$,*
- *and an algebraic assertion* exit *over V, called the* exit condition*.*

A transition τ is said to be deterministic *if it can be written in the form*

$$\bigwedge_j h_j(V) = 0 \ \wedge \ \bigwedge_i x_i' g_i(V) - f_i(V) = 0,$$

where every $x_i' \in V'$ occurs exactly once and $\neg\mathsf{exit} \models g_i(V) \neq 0$. For every i and j, the polynomial h_j is called guard *and the polynomial $x_i' g_i(V) - f_i(V)$ is called* update*: f_i is its* numerator *and g_i its* denominator*. The polynomial lasso program L is called* pseudo-deterministic *if all its transitions $\tau \in \mathcal{T}$ are deterministic.*

Lassos with solely deterministic transitions can have overlapping guards, hence the choice of transitions may be non-deterministic even in a pseudo-deterministic polynomial lasso program. Due to the nature of imperative languages, pseudo-deterministic lassos possess a specific interest to us.

Definition 6 (Semantics of a Lasso Program). *Let $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ be a polynomial lasso program. An* execution of *L is a (potentially infinite) sequence $\sigma = \nu_0\nu_1 \ldots$ where $\nu_i : V \to \mathbb{R}$ is a valuation on the variables V such that*

1. $\nu_0 \models \mathsf{stem}$
2. *For all $i \geq 0$ there is a $\tau \in \mathcal{T}$ such that $\tau(\nu_i, \nu_{i+1})$.*
3. $\nu_i \models \mathsf{exit}$ *iff it is the last element in σ.*

Example 1 (Running example). Consider the imperative program and its lasso representation L shown in Figure 1. L is a pseudo-deterministic lasso program since τ is a deterministic transition with the two update polynomials $y' - y + 1$ and $s' - s - x_0$ and no guards. An execution of L is $\sigma = \nu_0\nu_1$ where

$$\begin{aligned} \nu_0 &: \ x_0 \mapsto 3 \ y_0 \mapsto 1 \ y \mapsto 1 \ s \mapsto 0, \\ \nu_1 &: \ x_0 \mapsto 3 \ y_0 \mapsto 1 \ y \mapsto 0 \ s \mapsto 3. \end{aligned}$$

```
procedure product($x_0$, $y_0$):
    $s$ := 0;
    $y$ := $y_0$;
    while ($y \neq 0$):
        $s$ := $s + x_0$;
        $y$ := $y - 1$;
    return $s$;
```

Lasso program $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$:

$$\begin{aligned} V &= \{x_0, y_0, y, s\}, \\ \mathsf{stem} &\equiv s = 0 \land y = y_0, \\ \tau &\equiv y' = y - 1 \land s' = s + x_0, \\ \mathsf{exit} &\equiv y = 0, \\ \mathcal{T} &= \{\tau\} \end{aligned}$$

Fig. 1. An example imperative code and its representation as a polynomial lasso program (see Example 1). The program performs a multiplication by repeated addition.

Definition 7 (Correctness). *Let $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ be a polynomial lasso program and let* post *be an algebraic assertion over V. The lasso L is said to be* (partially) correct with respect to the post condition post *if for every finite execution σ of L, the last valuation in σ is a model of* post. *L is* totally correct with respect to post *if it is partially correct with respect to* post *and it is terminating, i.e., there are no infinite executions of L.*

Definition 8 (Invariant). *Let $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ be a polynomial lasso program. A polynomial $p \in \mathbb{R}[V]$ is called an* (inductive) invariant *of a transition $\tau \in \mathcal{T}$ if*

1. $\mathsf{stem} \models p(V) = 0$ *and*
2. $p(V) = 0 \land \tau(V, V') \land \neg\mathsf{exit} \models p(V') = 0$.

The polynomial p is called an (inductive) invariant of L *if it is an invariant of all transitions $\tau \in \mathcal{T}$.*

It is easily shown by means of induction that if p is an invariant of a lasso L, then for every execution σ of L and every valuation $\nu \in \sigma$, we have $\nu \models p = 0$.

Example 2. Example 1 calculates the product s of the two input values x_0 and y_0 by repeated addition. The polynomial lasso program L is partially correct with respect to the post condition $s = x_0 y_0$ and it is easy to check that $s + x_0 y - x_0 y_0 = 0$ is an invariant of L.

5 Polynomial Loop Invariants

In this section, we extend the approach for discovering loop invariants for polynomial lasso programs introduced by Sankaranarayanan, Sipma and Manna [1]. We define a weakened form of what they call *polynomial consecution.* We prove that under some restrictions, this is a complete approach for invariants over the complex numbers. The results established in this section will then be applied to program synthesis in section 6.

The first lemma relieves us in certain cases from the potentially very expensive computation of a Gröbner basis for the loop transitions. Specifically, for a deterministic transition τ, division by the Gröbner basis of τ is equivalent to substitution of the primed variables according to the update statements.

Lemma 3. *Let τ be a deterministic transition with at most one guard polynomial h and updates $x'_i - f_i(V)$ that have denominator* 1*. If $x'_i \succ x_j$ in the monomial ordering for all i and j, then the set $G = \{h(V)\} \cup \{x'_i - f_i(V) \mid 1 \leq i \leq n\}$ is a Gröbner basis of the ideal $\langle\tau\rangle$.*

For the remainder of this paper, let $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ be a fixed pseudo-deterministic polynomial lasso program. We will now define a sufficient, and under some assumptions also necessary, condition for a template polynomial to be an invariant of L.

Definition 9 (Invariance Condition). *For each transition $\tau \in \mathcal{T}$, let q_τ be any common multiple of the denominators of the update statements of τ. (In particular, q_τ can be the product of all denominators.) Let Ψ be a template polynomial over (A, V) of total degree d. Let $s(V)$ be the generator of* exit *if it has only one generator and* 1 *otherwise. The* invariance condition $\mathrm{IC}(L, \Psi)$ of L for *Ψ is the conjunction of*

$$\mathrm{NF}_{\mathsf{stem}}(\Psi(V)) = 0, \ \textit{and}$$
$$\mathrm{NF}_\tau(q_\tau(V)^d \cdot s(V) \cdot \Psi(V')) = \Phi_\tau(V) \cdot \Psi(V), \ \textit{for all } \tau \in \mathcal{T},$$

where the polynomials Φ_τ are generic template polynomials over (B_τ, V) whose degrees are bounded by the result of the division $\mathrm{NF}_\tau(q_\tau(V)^d \cdot s(V) \cdot \Psi(V'))$ *and B_τ are new disjoint sets of template variables.*

The variables V and V' are universally quantified in the invariance condition, whereas the variables A and $(B_\tau)_{\tau\in\mathcal{T}}$ are existentially quantified. By the Zero Polynomial Theorem 1, the equations in the invariance condition hold for all valuations on $V \cup V'$ if and only if all the coefficients of the polynomials are identical to zero. Therefore the variables V and V' can be removed from the invariance condition yielding a constraint on the variables A and $(B_\tau)_{\tau\in\mathcal{T}}$.

Remark 1. The invariance condition is designed to allow completeness in a wide variety of cases. We provide some intuition for its components below, but for details the reader is referred to the proof of Theorem 4.

- The result of the division $\mathrm{NF}_\tau(q_\tau(V)^d \cdot s(V) \cdot \Psi(V'))$ may not yield $\Psi(V)$, but rather some multiple of $\Psi(V)$. Hence, we have the generic template polynomial Φ_τ in the invariance condition.
- If an update statement, say $x'_i g_i - f_i$, in τ contains a nontrivial denominator g_i, then we may not be able to remove x'_i from $\Psi(V')$ by division on τ. Since every monomial in $\Psi(V')$ contains at most d primed variables, therefore multiplying $\Psi(V')$ with the polynomial $q_\tau(V)^d$ guarantees that division by τ will eliminate all primed variables.
- When the exit condition $s(V) = 0$ holds, we do not need Ψ to be inductive. Hence, we use the product $\Psi(V') \cdot s(V)$, which encodes that Ψ holds in the next state *or* the exit condition is satisfied.

– If the exit condition is generated by more than one polynomial, we cannot use this trick for all generators, thus loosing completeness. For simplicity, we set $s = 1$ in those cases, but selecting one of the exit condition's generators as s will make the condition more complete (but also more complex).

Remark 2. The invariance condition in Definition 9 is more general than the condition used by Sankaranarayanan et al. [1]. They use the following inductiveness property:

$$\mathrm{NF}_\tau(\Psi(V')) - \lambda \cdot \mathrm{NF}_\tau(\Psi(V)) = 0,$$

where λ is a real-valued variable. This not only restricts Φ_τ to a template of degree 0, it also omits the additions we have discussed in Remark 1.

Example 3. In order to state the invariance condition for Example 1, we first fix a template polynomial Ψ over V. The general second-degree template polynomial over V is the following.

$$\begin{aligned}\Psi(V) = {} & a_0 x_0^2 + a_1 y_0^2 + a_2 y^2 + a_3 s^2 + a_4 x_0 y_0 + a_5 x_0 y + a_6 x_0 s \\ & + a_7 y_0 y + a_8 y_0 s + a_9 y s + a_{10} x_0 + a_{11} y_0 + a_{12} y + a_{13} s + a_{14}\end{aligned}$$

The invariance condition $\mathrm{IC}(L, \Psi)$ is given by the following equations.

$$\begin{aligned} 0 = {} & a_0 x_0^2 + (a_1 + a_2 + a_7) y^2 + (a_4 + a_5) x_0 y + a_{10} x_0 + (a_{11} + a_{12}) y + a_{14} \\ 0 = {} & (a_0 + a_3 + a_6 - b a_0) x_0^2 y + (a_1 - b a_1) y_0^2 y + (a_2 - b a_2) y^3 + (a_3 - b a_3) y s^2 \\ & + (a_4 + a_8 - b a_4) x_0 y_0 y + (a_5 + a_9 - b a_5) x_0 y^2 + (a_6 + 2 a_3 - b a_6) x_0 s y \\ & + (a_7 - b a_7) y_0 y^2 + (a_8 - b a_8) y_0 y s + (a_9 - b a_9) y^2 s \\ & + (a_{10} + a_{13} - a_9 - a_5 - b a_{10}) x_0 y + (a_{11} - a_7 - b a_{11}) y_0 y \\ & + (a_{12} - 2 a_2 - b a_{12}) y^2 + (a_{13} - a_9 - b a_{13}) y s + (a_{14} - b a_{14}) y \end{aligned}$$

Here, $\Phi_\tau(V) = b \cdot y$ is the generic template polynomial over $B_\tau = \{b\}$ of degree 0 multiplied with y, the generator of exit (for simplicity of presentation, we abstained from using a generic template polynomial for Φ_τ). By Theorem 1, these two equalities yield 21 equations which are linear after assigning a value to b. The assignment $\alpha : A \cup B_\tau \to \mathbb{R}$ given by the following table is a solution to the invariance condition $\mathrm{IC}(L, \Psi)$.

	b	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}
α	1	0	0	0	0	−1	1	0	0	0	0	0	0	0	1	0

This yields the loop invariant $\tilde{\alpha}(\Psi) = s + x_0 y - x_0 y_0$ from Example 2.

Theorem 3 (Soundness). *If $\alpha : A \cup \bigcup_{\tau \in \mathcal{T}} B \to \mathbb{R}$ is an assignment for the template variables that is a solution to the invariance condition* $\mathrm{IC}(L, \Psi)$*, then $\tilde{\alpha}(\Psi)$ is an invariant of L.*

Proof. $\mathrm{NF}_{\mathsf{stem}}(\tilde{\alpha}(\Psi)) = 0$, hence $\tilde{\alpha}(\Psi) \in \langle \mathsf{stem} \rangle$, and therefore $\mathsf{stem} \models \tilde{\alpha}(\Psi) = 0$ according to Lemma 1. By the premise,

$$q_\tau(V)^d \, s(V) \, \tilde{\alpha}(\Psi)(V') - \tilde{\alpha}(\Phi_\tau)(V) \, \tilde{\alpha}(\Psi)(V) \in \langle \tau \rangle$$

for all $\tau \in \mathcal{T}$, therefore $q_\tau(V)^d s(V)\tilde{\alpha}(\Psi)(V') \in \langle \tau, \tilde{\alpha}(\Psi)(V)\rangle$ by Lemma 2, and from Lemma 1 follows

$$\tau(V,V') \wedge \tilde{\alpha}(\Psi)(V) = 0 \models q_\tau(V)^d \cdot s(V) \cdot \tilde{\alpha}(\Psi)(V') = 0.$$

Since q_τ is a common multiple of denominators of updates in τ and $\neg\mathsf{exit}$ holds before any transition τ, it follows that $\neg\mathsf{exit} \models q_\tau(V) \neq 0$ by Definition 5. With $\neg\mathsf{exit} \models s(V) \neq 0$ we conclude that

$$\tau(V,V') \wedge \tilde{\alpha}(\Psi)(V) = 0 \wedge \neg\mathsf{exit} \models \tilde{\alpha}(\Psi)(V') = 0. \qquad \square$$

A criterion for the method's completeness is given by the following theorem. The Nullstellensatz is applicable only when one considers the theory of complex numbers, which in general admits a proper subset of loop invariants. Furthermore, the Nullstellensatz demands all involved ideals be radical ideals [18].

Theorem 4 (Completeness in $\mathbb{C}$). *Let $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ be a polynomial lasso program with the complex loop invariant*[1] *$p \in \mathbb{R}[V]$. If $\alpha : A \to \mathbb{R}$ is a valuation such that $\tilde{\alpha}(\Psi) = p$, then α can be extended to a solution to the invariance condition if the following additional premises are met.*

1. *The lasso L is pseudo-deterministic.*
2. *The ideal $\langle\mathsf{stem}\rangle$ and the ideal $\langle p\rangle$ are both radical ideals.*
3. *The ideal $\langle\mathsf{exit}\rangle$ is generated by a single polynomial $s \in \mathbb{R}[V]$.*
4. *The guard $h = 0$ of each transition $\tau \in \mathcal{T}$ is equivalent to True (i.e., h is 0).*
5. *The monomial ordering $\succ$ is lexicographic and $x_i' \succ x_j$ for all i, j.*

Proof. The polynomial p is a loop invariant of L, so by Definition 8,

$$\mathsf{stem} \models_{\mathbb{C}} p(V) = 0 \text{ and} \tag{1}$$

$$p(V) = 0, \tau(V,V'), \neg\mathsf{exit} \models_{\mathbb{C}} p(V') = 0 \text{ for all } \tau \in \mathcal{T}. \tag{2}$$

The ideal $\langle\mathsf{stem}\rangle$ is a radical ideal by Premise 2, so according to Hilbert's Nullstellensatz, Equation (1) implies $p \in \langle\mathsf{stem}\rangle$; and hence, α satisfies the first part of the invariance condition (IC).

To prove that α can be extended to satisfy the second part of IC, note that Equation (2), combined with Premise 3, yields

$$p(V) = 0, \tau(V,V') \models_{\mathbb{C}} s(V)p(V') = 0.$$

Using the Nullstellensatz, for some positive number k, we have

$$\big(q_\tau(V)^d s(V) p(V')\big)^k \in \langle p, \tau\rangle.$$

Since h is 0, normalizing by τ is equivalent to replacing primed variables using the update expressions in τ, and hence,

$$s(V)^k r(V)^k \in \langle p, \tau\rangle, \quad \text{where } r(V) := \mathrm{NF}_\tau(q_\tau(V)^d p(V'))$$

[1] The assertions of Definition 8 hold in the theory of the complex numbers.

Note that $r(V)$ has no prime variables since Premise 5 ensures all prime variables are greater with respect to the monomial ordering $\succ$ than the unprimed variables. Therefore, $s(V)^k r(V)^k \in \langle p, \tau \rangle \cap \mathbb{R}[V]$. Now, there are two cases.
(Case 1): $\langle p, \tau \rangle \cap \mathbb{R}[V] = \langle p \rangle$. Then, it follows that $s(V)^k r(V)^k \in \langle p \rangle$. Since $\langle p \rangle$ is a radical ideal, we can infer $s(V)r(V) \in \langle p \rangle$ and hence $\mathrm{NF}_\tau(q_\tau(V)^d s(V) p(V'))$ is a multiple of p. Hence, second part of IC is satisfied.
(Case 2): $\langle p, \tau \rangle \cap \mathbb{R}[V] \neq \langle p \rangle$. This is possible only if some multiple of the denominators rewrites to 0 by p. Hence, $p = 0$ implies $s(V) = 0$ (since $s \neq 0$ implies that denominators are nonzero). Since $\langle p \rangle$ is a radical ideal, it follows $s \in \langle p \rangle$, and hence $s(V)r(V) \in \langle p \rangle$ — as in (Case 1) above. □

It is important to emphasize that the generic template polynomial for the invariant must have a sufficiently large degree to be able to specialize to the loop invariant. This is presumed in the completeness statement. We will now discuss the other premises of Theorem 4.

Premise 1 ensures that the division of $\Psi(V')$ on a transition τ removes all primed variables, since we multiplied with $q_\tau(V)^d$ in the invariance condition. Premise 2 is a requirement by Hilbert's Nullstellensatz. In order to write a disjunction of exit and a polynomial equality as a product, exit must have a single generator; this is stated in Premise 3. We will discuss relaxing Premise 4 below. Finally, Premise 5 assures that primed variables are eliminated first, leaving only unprimed variables in appropriate cases. This is relevant because the right hand side $\Phi_\tau(V) \cdot \Psi(V)$ in the invariant condition contains only unprimed variables.

Remark 3. We can generalize the completeness result to also include the case when guards of transitions are nontrivial and when a conjunction $p_1 = 0 \wedge p_2 = 0$ is an inductive invariant, but neither $p_1 = 0$ nor $p_2 = 0$ by itself is an inductive invariant. This requires generalizing the second part of the invariance condition. Let Ψ_1 and Ψ_2 be the templates whose instantiation gives p_1 and p_2 respectively. Then, for all τ in $\mathcal{T}$, and for $i = 1, 2$,

$$\mathrm{NF}_\tau(q_\tau(V)^d \cdot s(V) \cdot \Psi_i(V')) = \Phi_1(V) \cdot \Psi_1(V) + \Phi_2(V) \cdot \Psi_2(V) + \Phi_3(V) \cdot h_\tau(V)$$

Note that Φ_1, Φ_2, Φ_3 are different templates for different τ's and different i's. As before, the degrees of the templates are bounded by the degree of the left-hand side, and d is the total degree of Ψ_i. In the completeness theorem, we can now drop Premise 4, but replace Premise 2 by the following generalization:

2 The ideal stem, and for all τ, the ideals $\langle p_1, p_2, h_\tau \rangle$, where $h_\tau = 0$ is the guard of τ, are radical ideals. Moreover, $\{p_1, p_2, h_\tau\}$ is a GB of $\langle p_1, p_2, h_\tau \rangle$.

The proof of the new completeness claim is a natural generalization of the proof of Theorem 4 above. □

Besides the five restrictions of Theorem 4, completeness does not extend to the field of real numbers due to the requirements of Hilbert's Nullstellensatz. The underlying problem is illustrated by the following example.

Example 4. The formula $\varphi \equiv x_1^2 + x_2^2 = 0$ has $x_1 = x_2 = 0$ as its only solution over the reals. However, $x_1, x_2 \notin \langle x_1^2 + x_2^2 \rangle$, although $\langle x_1^2 + x_2^2 \rangle$ is a radical ideal and $\varphi \models_{\mathbb{R}} x_1 = 0, x_2 = 0$.

Alternatively, we could formulate our results using *real radical ideals* [19].

Because the invariance condition in general is a non-linear constraint, solving it might be very difficult. General approaches for solving non-linear constraints have worst case space requirements that are doubly exponential in the size of the input. However, non-linear constraint solving is an active field of research and recently there have been some promising efforts to take the practical cases away from their DOUBLE-EXPSPACE worst-case complexity bound [20].

Another approach for solving the invariance condition stems from the observation that the invarianc condition becomes linear if an assignment for the template variables $(B_\tau)_{\tau \in \mathcal{T}}$ is given. One could use heuristics to find this assignment. For instance, practical experience suggests that if a solution to a variable $b \in B_\tau$ is $\lambda_b \in \mathbb{R}$, then the factor $(b - \lambda_b)$ occurs somewhere in the invariance condition. Using factors in the former form as an initial guess for the variables $(B_\tau)_{\tau \in \mathcal{T}}$ linearizes the equations and thus enables quick discovery of a solution in some cases.

In the special case that $\Phi_\tau(V) := \lambda$ is degree 0 (also called constant consecution), λ can be found as an eigenvalue of an appropriate transformer constructed by interpreting bounded degree polynomials as finite-dimensional vector spaces [21].

6 Synthesis

The technique for finding a loop invariant using the invariance condition established in the previous section will now be used for program synthesis. Given a polynomial lasso program, some transition updates can be parameterized by replacing them with template polynomials. The synthesis process will try to find a valuation of these template variables while respecting some post condition. The following definition formalizes this concept.

Definition 10 (Synthesis Problem). *A* synthesis problem $S = (C, L, \mathsf{post})$ *consists of*

- *a set of* synthesis variables C*,*
- *a polynomial lasso program* $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ *where* stem *and* $\tau \in \mathcal{T}$ *contain template polynomials over* (C, V)*, and*
- *a* post condition *in form of an algebraic assertion* post *over* V*.*

A solution to the synthesis problem S *is a valuation* $\alpha : C \to \mathbb{R}$ *such that the lasso* $L_\alpha = (V, \tilde{\alpha}(\mathsf{stem}), \tilde{\alpha}(\mathcal{T}), \mathsf{exit})$ *is partially correct with respect to the post condition* post.

Example 5. Transforming L from Example 1 to L' by changing the transition τ to

$$y' = y - 1 \land s' = c_1 x_0 + c_2 y_0 + c_3 y + c_4 s + c_5$$

gives rise to a synthesis problem $S = (C, L', \mathsf{post})$ for $C = \{c_1, c_2, c_3, c_4, c_5\}$ and $\mathsf{post} \equiv s = x_0y_0$. A solution to S is $\alpha : c_1 \mapsto 1, c_2 \mapsto 0, c_3 \mapsto 0, c_4 \mapsto 1, c_5 \mapsto 0$ since $L_\alpha = L$ and L is partially correct with respect to post according to Example 2.

Our approach for solving the synthesis problem is based on the technique from the previous section. We will prove the partial correctness of the synthesized lasso program. The following lemma states that synthesized polynomial lasso program will be partially correct.

Lemma 4 (Synthesis Solution). *Let $S = (C, L, \mathsf{post})$ be a synthesis problem, $\alpha : C \to \mathbb{R}$ be a valuation on the synthesis variables, and let p be an invariant for L_α. If $p = 0 \wedge \mathsf{exit} \models \mathsf{post}$, then L_α is partially correct with respect to* post*, i.e., α is a solution to S.*

Proof. Let $\sigma = \nu_0 \dots \nu_k$ be a finite execution of L_α. According to the assumption, p is an invariant of L_α, so by Definition 8, $\nu_i \models p = 0$ for all $0 \le i \le k$. By Definition 6, $\nu_k \models \mathsf{exit}$, therefore $\nu_k \models p = 0 \wedge \mathsf{exit}$. According to the assumption, this implies $\nu_k \models \mathsf{post}$, which proves the correctness of L_α. □

To find a valuation for the synthesis variables, we define a *synthesis condition.* The synthesis condition will constrain the synthesis variables so that existence of a loop invariant p that implies the post condition is guaranteed; that is,

$$p = 0 \wedge \mathsf{exit} \models \mathsf{post}. \tag{3}$$

If $\mathsf{post} = \bigwedge_i \mathsf{post}_i = 0$, then the above is implied by $\mathsf{post}_i \in \langle p, \mathsf{exit}\rangle$ by Lemma 1. However, computing the Gröbner basis with respect to a template polynomial for p is extremely inefficient and potentially involves a huge number of case splits. But according to Lemma 2, we can equivalently write

$$\mathsf{post}_i - tp \in \mathsf{exit}, \tag{4}$$

for some unknown $t \in \mathbb{R}[V]$. This enables us to rewrite (3) in a way that only involves computing the Gröbner basis for non-template polynomials.

Example 6. Let S be the synthesis problem from Example 5. We use the loop invariant $p = s + x_0y - x_0y_0$ from Example 2 in Lemma 4 to show that α is a solution to S by checking

$$s + x_0y - x_0y_0 = 0 \wedge y = 0 \models s = x_0y_0,$$

or instead that for $t = 1$,

$$(s - x_0y_0) - t(s + x_0y - x_0y_0) \in \mathsf{exit}.$$

Definition 11 (Synthesis Condition). *Let $S = (C, L, \bigwedge_{i=1}^m \mathsf{post}_i(V){=}0)$ be a synthesis problem, let Ψ be a template polynomial over (A, V) and for all $0 \le i \le m$, let Ω_i be a template polynomial over (D_i, V). The* synthesis condition, $\mathrm{SC}(S, \Psi, \{\Omega_i \mid 0 \le i \le m\})$*, of S is the formula*

$$\mathrm{IC}(L, \Psi) \wedge \bigwedge_i \mathrm{NF}_{\mathsf{exit}}(\mathsf{post}_i(V) - \Omega_i(V)\Psi(V)) = 0$$

Following the same argument as for the invariant condition, the synthesis condition simplifies to a conjunction of non-linear equations in the variables $A \cup C \cup (\bigcup_{\tau \in \mathcal{T}} B_\tau) \cup (\bigcup_{i=1}^{m} D_i)$. Utilizing an SMT solver, a solution to this constraint can be obtained that is then used to instantiate the template polynomials in the loop invariant and polynomial lasso program. According to the next theorem, this yields a correct program instance.

Motivated by Example 6, we may set $\Omega_i = 1$ in the synthesis condition. In this case the constraint $\mathrm{NF}_{\mathsf{exit}}(\mathsf{post}_i(V) - \Psi(V)) = 0$, the synthesis condition's constraint corresponding to the post condition, is linear. Using this observation we can use linear methods to eliminate some variables from the constraint system, thus simplifying it. The same trick also applies to the constraint $\mathrm{NF}_{\mathsf{stem}}(\Psi(V)) = 0$ in the invariance condition if the program stem does not contain any synthesis variables C.

Because the coefficients of some of the polynomials in L contain template variables, special care must be taken when computing a Gröbner basis for stem or $\tau \in \mathcal{T}$. Every division by some term containing a variable demands a case split on whether this term evaluates to zero. One way of circumventing this problem is to compute a Gröbner basis where the underlying algebraic structure for polynomial coefficients is the ring of parameter polynomials $\mathbb{R}[A]$. This requires a slightly modified division algorithm [22,23].

Theorem 5 (Synthesis Soundness). *If $\alpha : A \cup (\bigcup_{\tau \in \mathcal{T}} B_\tau) \cup C \cup (\bigcup_i D_i) \to \mathbb{R}$ is an assignment for the template variables that models the synthesis condition, then L_α is partially correct with respect to the post condition* post *and $\tilde{\alpha}(\Psi)$ is an invariant of L_α.*

Proof. By Theorem 3, $\tilde{\alpha}(\Psi)$ is an invariant of L_α. By definition, $\mathsf{post}_i(V) - \tilde{\alpha}(\Omega_i)(V)\tilde{\alpha}(\Psi)(V) \in \langle \mathsf{exit} \rangle$, therefore $\mathsf{post}_i \in \langle \tilde{\alpha}(\Psi), \mathsf{exit} \rangle$ for all i according to Lemma 2. By Lemma 1, $\tilde{\alpha}(\Psi) \wedge \mathsf{exit} \models \mathsf{post}_i$ for all i, hence $\tilde{\alpha}(\Psi) \wedge \mathsf{exit} \models \mathsf{post}$. Lemma 4 ensures that this implies that L_α is partially correct. □

The synthesis process is not complete, even when the restrictions of Theorem 4 hold. The reason for this is the polynomial t in (4): we are using templates Ω_i for t, but a priori we have no upper bound on the degree of t. In practice, a template of degree 0 might be sufficient, as in our examples (see section 8).

7 Termination

A solution to the synthesis problem guarantees partial correctness of the synthesized program; however, termination is not guaranteed. Even if the synthesized program terminates, it might be highly inefficient, going through unnecessarily many loop iterations.

Example 7. If one extends L' from Example 5 to L'' by changing τ to

$$y' = c_6 y + c_7 \wedge s' = c_1 x_0 + c_2 y_0 + c_3 y + c_4 s + c_5$$

this yields a synthesis problem $S' = (C', L'', \mathsf{post})$ with $C' = C \cup \{c_6, c_7\}$. Possible solutions to S' include the valuations $\alpha_\lambda : c_1 \mapsto \lambda, c_2 \mapsto 0, c_3 \mapsto 0, c_4 \mapsto 1, c_5 \mapsto 0, c_6 \mapsto 1, c_7 \mapsto -\lambda$ for all $\lambda \in \mathbb{R}$.

If λ is small, the program needs more iterations for the same input, and if λ is zero, L_α will not terminate at all.

In order to address this, the synthesis condition can be augmented with a series of test cases, predefined input-output pairs that explicitly state the transitions required to compute them.

Definition 12 (Test Case). *Let (C, L, post) be a synthesis problem where $L = (V, \mathsf{stem}, \mathcal{T}, \mathsf{exit})$ is a pseudo-deterministic polynomial lasso program containing template variables C. A* test case $t = (\nu_0, \nu, \tau_1 \dots \tau_k)$ *consists of two V-valuations ν_0 and ν corresponding to the initial and final state respectively such that $\nu \models \mathsf{exit}$, as well as a finite sequence of transitions $\tau_1, \dots, \tau_k \in \mathcal{T}$. A solution α to a synthesis problem S is said to* adhere to the test case t *if, for $\nu_i = \tau_i \circ \dots \circ \tau_1(\nu_0)$, the sequence $\sigma = \nu_0 \nu_1 \dots \nu_k$ is an execution of L_α and $\nu_k = \nu$.*

Lemma 5. *Let $S = (C, L, \mathsf{post})$ be a synthesis problem with solution α and let $t = (\nu_0, \nu, \tau_1 \dots \tau_k)$ be a test case. If*

$$\nu_0 \models \alpha(\mathsf{stem}), \tag{5}$$

$$\nu = \alpha(\tau_k) \circ \dots \circ \alpha(\tau_1)(\nu_0), \tag{6}$$

$$\nu_i \not\models \mathsf{exit} \textit{ for } 0 \le i \le k-1, \textit{ and} \tag{7}$$

$$\nu_k \models \mathsf{exit} \tag{8}$$

then L_α adheres to the test case t.

Proof. $\sigma = \nu_0 \nu_1 \dots \nu_k$ for $\nu_i = \alpha(\tau_i) \circ \dots \circ \alpha(\tau_1)(\nu_0)$ is by construction an execution of L_α according to Definition 6. From (6) follows that $\nu_k = \nu$. □

If we add the equations (5), (6), (7) and (8) to the synthesis condition for every given test case, then by Lemma 5 any solution to these constraints will yield a solution to S that adheres to the test cases.

Example 8. Consider the synthesis problem S' from Example 7. The execution σ from Example 1 gives rise to the test case $t = (\nu_0, \nu_1, \tau)$, which by Lemma 5 adds the following additional constraints on the synthesis condition.

$$\begin{array}{lll} 1 = 1 & 0 = 1c_6 + c_7 & 1 \neq 0 \\ 0 = 0 & 3 = 3c_1 + 1c_2 + 1c_3 + 0c_4 + c_5 & 0 = 0 \end{array}$$

The valuation α_1 is the only one of the valuations α_λ given in Example 7 that models these two equations (however, it is not the only possible solution). L_{α_1} is a terminating lasso program for positive integers y_0.

In theory, if it is possible to synthesize a terminating program, then there exists a finite set of test cases that will guarantee that a terminating lasso is synthesized.

Theorem 6. *Let $S = (C, L, \mathsf{post})$ be a synthesis problem. If there is a solution α to S such that L_α is terminating then there is a finite set of test cases Σ such that any solution β of S which adheres to all test cases $t \in \Sigma$ is terminating.*

Proof. Let $\Sigma = \{t_0, t_1, \ldots\}$ be the test cases to all possible executions of L_α, and assume Σ is infinite (otherwise there is nothing to show). Each test case $t \in \Sigma$ corresponds to a polynomial assertion over the variables C by (5) and (6). This assertion constrains possible assignments of C. For every $i \geq 0$, let $\Sigma_i = \{t_0, \ldots, t_i\} \subset \Sigma$ be an ascending chain of finite subsets of Σ and let I_i be the ideal generated by the assertions from the test cases of Σ_i. It is clear that $\Sigma_i \subset \Sigma_{i+1}$, and hence $I_i \subseteq I_{i+1}$. By the *Ascending Chain Condition* [18], the ascending chain of ideals $I_0 \subseteq I_1 \subseteq \ldots$ must become stationary for some integer k, meaning $I_k = I_i$ for all $i \geq k$. This implies that the finite set of test cases Σ_k corresponds to the same ideal as Σ_i for $i \geq k$ and hence they have the same solution (set of assignments) for C. As a consequence, any solution β to S adhering to the test cases from Σ_k will enforce that σ is an execution of L_β iff it is an execution of L_α. □

While Theorem 6 assures that under any circumstances, a finite set of test cases Σ suffices to force a useful solution from the synthesis problem, no upper bound to the cardinality of Σ is given.

In theory, this provides us with two powerful approaches of generating polynomial lasso programs, given an a priori bound on the number of program variables V. Both involve creating a polynomial lasso program with generic template polynomials as updates and guards.

1. Specify a (large) number of test cases. Ideally, these test cases can be automatically generated in some sophisticated way that ensures that they are not too redundant.
2. Provide a post condition and a complexity guess. Using the complexity guess, a terminating skeleton of the synthesis problem is generated using counter variables. The post condition provides a statement regarding the program's purpose.

Needless to say, both approaches create very large synthesis conditions that are unlikely to be handled automatically by present-day non-linear solvers, but this can change, especially for small program fragments, as technology develops.

8 Experimental Evaluation

We implemented our method in Haskell and used `nlsat` [20][2] to solve the non-linear constraints. To evaluate the practicability and scalability of our method, we ran it on a few selected examples which are listed in Table 1 together with a short description. Each example translates to a pseudo-deterministic polynomial lasso program.

[2] As implemented in z3 version 4.3.1. `http://z3.codeplex.com/`

Table 1. Example programs used to perform verification and synthesis experiments described in Table 2

name	*description*
`product`	multiplication of two integers by repeated addition (see Figure 1 and Example 1)
`productS`	`product` with synthesis of one update statement (see Example 5)
`productSY`	`product` with synthesis of the loop body, including the termination-critical variable y (see Example 7)
`product2`	`product` with reciprocal y
`product2S`	`product2` with synthesis of one update statement
`gcd_lcm`	greatest common denominator and least common multiple of two integers [1]
`gcd_lcmS`	`gcd_lcm` with synthesis of two update statements
`div_mod`	integer division with remainder [24]
`div_modS`	`div_mod` with synthesis of the complete loop body with linear updates
`root2`	integer square root [24]
`root2S`	`root2` with synthesis of the stem and one update statement
`squareS`	square of an integer synthesized from a terminating skeleton with linear assignments
`cubeS`	cube of an integer synthesized from a terminating skeleton with linear assignments

Table 2. Experimental results showing the time required to verify/synthesize various example programs, along with the size of the non-linear constraints solved in the process

name	#C	deg	#A	#vars	#tc	*constraints time (s)*	*solver time (s)*
`product`	0	2	15	20	0	0.55	0.02
`productS`	5	2	15	25	0	1.47	0.01
`productSY`	7	2	15	27	2	3.39	0.02
`product2`	0	3	35	50	0	39.23	128.24
`product2S`	5	3	35	55	0	200.20	24.46
`gcd_lcm`	0	2	28	42	0	11.85	0.02
`gcd_lcmS`	10	2	28	52	0	17.01	0.01
`div_mod`	0	2	15	16	0	0.62	0.01
`div_modS`	10	2	15	26	5	10.03	0.03
`root2`	0	2	15	25	0	2.80	4.52
`root2S`	9	2	15	34	0	3.80	0.02
`squareS`	6	2	10	20	0	0.56	0.00
`cubeS`	14	3	35	54	0	90.88	41.05

Table 2 contains the experiment's results. We list the program name together with the number of synthesis variables (#C), the degree of the loop invariant's generic template polynomial (deg), the number of its template variables (#A), the total number of variables in the generated constraint (#vars), the number of test cases used (#tc), the time to generate the constraint in seconds (constraints time) and the running time of the SMT solver in seconds (solver time). Our test system was a computer with eight AMD Opteron 8220 2.80GHz CPUs and 32GB RAM.

While the synthesis process is very fast for small examples, the non-linear constraint solver becomes the bottleneck in medium-sized problems (`product2`, `product2S` and `cubeS` use generic templates of degree 3): solving non-linear constraints scales poorly with the number of variables involved. Test cases might help mitigate this issue by significantly reducing the solution space.

9 Conclusion

We presented a method for synthesizing polynomial programs. This method is based on the discovery of non-linear loop invariants that prove the program's correctness. We generate a *synthesis condition*, a non-linear constraint whose solution is the synthesized polynomial lasso program and a loop invariant. We extended existing methods for non-linear invariant generation and provided a completeness criterion (Theorem 4). If we synthesize update statements of variables that occur in the exit condition, termination becomes a concern. We showed that we can utilize a finite set of test cases to restrict the solution space to terminating lassos (Theorem 6).

Using a benchmark of small examples, we showed that our method is applicable for the synthesis of small programs, as well as parts of medium-sized ones. A resource bottleneck is the non-linear constraint solver. As the solving of non-linear constraints is an active area of research, we expect that our technique will become more effective as non-linear solvers improve.

We assumed that the programs' variables take values in the set of reals $\mathbb{R}$, but since Gröbner bases are computable over rings [22,23], our method can also be applied to the integers $\mathbb{Z}$ or the finite ring of machine integers $\mathbb{Z}_{2^w}$ (see also [10]). Future work could also consider the question of how this method can be improved to handle inequalities.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback.

References

1. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL (2004)
2. Solar-Lezama, A., Tancau, L., Bodík, R., Saraswat, V., Seshia, S.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
3. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011)
4. Tiwari, A.: An algebraic approach for the unsatisfiability of nonlinear constraints. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 248–262. Springer, Heidelberg (2005)
5. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)

6. Stengle, G.: A Nullstellensatz and a Positivstellensatz in semialgebraic geometry. Math. Ann. 207 (1974)
7. Parrilo, P.A.: Semidefinite programming relaxations for semialgebraic problems. Mathematical Programming Ser. B 96(2) (2003)
8. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Inf. Process. Lett. 91(5) (2004)
9. Karr, M.: Affine relationships among variables of a program. Acta Informatica 6, 133–151 (1976)
10. Seidl, H., Flexeder, A., Petter, M.: Analysing all polynomial equations in $\mathbb{Z}_{2^w}$. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 299–314. Springer, Heidelberg (2008)
11. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial loop invariants: Algebraic foundations. In: ISSAC. ACM (2004)
12. Rodríguez-Carbonell, E., Kapur, D.: Program verification using automatic generation of invariants. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 325–340. Springer, Heidelberg (2005)
13. Colón, M.A.: Polynomial approximations of the relational semantics of imperative programs. Sci. Comput. Program. 64(1) (2007)
14. Cachera, D., Jensen, T., Jobin, A., Kirchner, F.: Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 58–74. Springer, Heidelberg (2012)
15. Bradley, A.R., Manna, Z., Sipma, H.B.: Polyranking for polynomial loops (2005)
16. Colón, M.A.: Schema-guided synthesis of imperative programs by constraint solving. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 166–181. Springer, Heidelberg (2005)
17. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL 2010, pp. 313–326 (2010)
18. Cox, D., Little, J., O'Shea, D.: Ideals, Varieties and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer (1991)
19. Neuhaus, R.: Computation of real radicals of polynomial ideals – ii. Journal of Pure and Applied Algebra 124(1-3), 261–280 (1998)
20. 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)
21. Rebiha, R., Matringe, N., Moura, A.V.: Endomorphisms for non-trivial non-linear loop invariant generation. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 425–439. Springer, Heidelberg (2008)
22. Adams, W.W., Loustaunau, P.: An Introduction to Gröbner Bases. American Mathematical Society (1994)
23. Bachmair, L., Tiwari, A.: D-bases for polynomial ideals over commutative noetherian rings. In: Comon, H. (ed.) RTA 1997. LNCS, vol. 1232, pp. 113–127. Springer, Heidelberg (1997)
24. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)

Policy Iteration-Based Conditional Termination and Ranking Functions

Damien Massé

Univ. de Brest, UMR 6285, Lab-STICC, F-29200 Brest, France
damien.masse@univ-brest.fr

Abstract. Termination analyzers generally synthesize ranking functions or relations, which represent checkable proofs of their results. In [23], we proposed an approach for conditional termination analysis based on abstract fixpoint computation by policy iteration. This method is not based on ranking functions and does not directly provide a ranking relation, which makes the comparison with existing approaches difficult. In this paper we study the relationships between our approach and ranking functions and relations, focusing on extensions of linear ranking functions. We show that it can work on programs admitting a specific kind of segmented ranking functions, and that the results can be checked by the construction of a disjunctive ranking relation. Experimental results show the interest of this approach.

1 Introduction

Many approaches have been proposed to prove that a program terminates. Most techniques rely on the construction of ranking functions [13,24] or ranking relations as transition invariants [25]. Ranking functions and relations, like invariants in safety analysis, offer a checkable result, not directly related to the technique used to construct them. Similarly, a conditional termination analysis [7], i.e. an analysis which determines the set of terminating states, is expected to return not only the a set of terminating states, but an associated ranking function or relation.

In [23], we proposed to use policy iteration techniques in order to analyze conditional termination. Policy iteration (or strategy iteration) [10,19] has been developed as an alternative to the classical widening/narrowing techniques used for safety analysis by abstract interpretation on numerical programs. Applied on conditional termination, it can be used to compute, using linear optimization algorithms, an overapproximation of the (potentially) non-terminating states, hence proving the termination of the other states. However, it does not directly provide any ranking function or relation.

In this paper, we examine the relationships between our approach and ranking functions synthesis. We focus especially on disjunctive linear ranking relations and segmented linear ranking functions. We show the construction of a disjunctive ranking relation from the analysis, as well as the existence of a segmented linear ranking function. Reciprocally, we show that programs admitting some restricted form of segmented ranking function can be analyzed by policy iteration. We complete these results by a practical experiments on a prototype analyzer.

2 Notations

Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ be a tuple of n variables. We denote any element of $\mathbb{R}^{\boldsymbol{x}}$ either by $(x_1 = v_1; \ldots; x_n = v_n)$ or as a column vector $\boldsymbol{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$ of $\mathbb{R}^n$. When used as a matrix block, $\boldsymbol{x}$ represents the column vector $\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$. Linear forms on $\mathbb{R}^{\boldsymbol{x}}$ are denoted either as a expression $(\ c_1x_1 + \ldots + c_nx_n\ )$ or as a row vector $(c_1\ \ldots\ c_n)$. Similarly, $m \times n$-matrices $M \in \mathbb{R}^{m\times n}$ may be denoted as a m-tuple of linear forms on $\mathbb{R}^{\boldsymbol{x}}$. Given a matrix M, we denote by M^T its transpose.

3 Affine Programs and Semantics

3.1 Programs

For the sake of simplicity, we consider in this paper programs with only one program point. Hence, $\boldsymbol{x}$ being the set of (real) variables, a program is a pair $(I, \mathbf{T})$ where $I \subseteq \mathbb{R}^{\boldsymbol{x}}$ is the set of initial states, and $\mathbf{T} \subseteq \wp(\mathbb{R}^{\boldsymbol{x}} \times \mathbb{R}^{\boldsymbol{x}})$ describes the transitions, each transition defining a relation between the values of the variables before and after the transition.

Since we are interested in affine programs, I and $\mathbf{T}$ will be defined by linear constraints, that is:

1. I is a set of linear constraints on $\boldsymbol{x}$
2. Each transition of $\mathbf{T}$ is described as a set of linear constraints on $\boldsymbol{x}, \boldsymbol{x'}$ where $\boldsymbol{x'}$ represents the variables after the transition.

In our framework, I can be used to compute an overapproximation of the reachable states. However, for simplicity we will assume that all states are reachable, and I will not be used.

Example 1. We consider the program A of Fig 1. $\mathbf{T} = \{t_1, t_2\}$ is represented with:

$$t_1 : \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ a' \\ b' \end{pmatrix} \begin{matrix} \leq \\ \leq \\ = \\ = \end{matrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

Program A:
```
1: while $a \geq 0$ do
2:     $a \leftarrow a + b$
3:     if $b \geq 0$ then
4:         $b \leftarrow -b - 1$
5:     else
6:         $b \leftarrow -b$
7:     end if
8: end while
```

Program B:
```
1: while $x \leq 100$ do
2:     if (*) then
3:         $x \leftarrow -2x + 2$
4:     else
5:         $x \leftarrow -3x - 2$
6:     end if
7: end while
```

Fig. 1. Two affine programs. a and b are integer variables, x is a real variable, and (*) is a non-deterministic choice.

$$t_2 : \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ a' \\ b' \end{pmatrix} \begin{matrix} \leq \\ \leq \\ = \\ = \end{matrix} \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \end{pmatrix}$$

Note that we replace $b < 0$ by $b \leq -1$ in the representation, since b is an integer variable. Our approach deals mainly with real variables, and using strict constraints (or floating-point computations) requires technical considerations which are outside the scope of this paper.

In the following, we will represent a transition as a pair $(Q, \boldsymbol{q})$ where Q is a matrix and $\boldsymbol{q}$ is a vector such that the associated constraints are:

$$Q \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x'} \end{pmatrix} \leq \boldsymbol{q}$$

When the program has only one transition, the program is called a *Linear Simple Loop* (LSL). Termination of Linear Simple Loop is well known to be a decidable problem (at least for linear assignments) [29]. However, deciding conditional termination of LSL is more complex [4], and not always possible [16].

3.2 Concrete Semantics

To each program P is associated a transition relation $\tau \subseteq \mathbb{R}^{\boldsymbol{x}} \times \mathbb{R}^{\boldsymbol{x}}$. This relation can be used to construct a trace-based semantics to prove termination [12]. However, we propose to use here a state based backward semantics:

Proposition 1. *The set $\mathcal{S}$ of states starting an infinite execution trace is equal to:*

$$\mathcal{S} = \text{gfp}\, \lambda Y.\text{pre}(Y)$$

where $\text{pre} \in \wp(\mathbb{R}^{\boldsymbol{x}}) \to \wp(\mathbb{R}^{\boldsymbol{x}})$ *is the predecessor predicate transformer:*

$$\begin{aligned} \text{pre}(Y) &= \{\boldsymbol{v} \in \mathbb{R}^{\boldsymbol{x}} \mid \exists \boldsymbol{v'} \in Y, (\boldsymbol{v}, \boldsymbol{v'}) \in \tau\} \\ &= \bigcup_{(Q,\boldsymbol{q}) \in \mathbf{T}} \{\boldsymbol{v} \in \mathbb{R}^{\boldsymbol{x}} \mid \exists \boldsymbol{v'} \in Y, Q \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{v'} \end{pmatrix} \leq \boldsymbol{q}\} \end{aligned}$$

If Y is a polyhedron characterized with a set of linear constraints $\boldsymbol{Ax} \leq \boldsymbol{c}$, pre$(Y)$ is a union of polyhedra, which can be computed using libraries like Apron [21]. However, even if pre(Y) is computable, $\mathcal{S}$ is not computable, since the fixpoint computation may not terminate.

As seen in [12], it is well-known that a ranking function can be defined from the iterates of the pre operator:

Proposition 2. *Let* $(S_i)_{i \in \mathbb{O}}$ *be the iterates of* $\mathrm{gfp}\, \lambda Y.\mathrm{pre}(Y)$ *(i.e.* $S_0 = \mathbb{R}^{\boldsymbol{x}}$, $S_{i+1} = \mathrm{pre}(S_i)$ *and, for all limit ordinal* l, $S_l = \cap_{i<l} S_i$*). Then the function* r *defined on* $\Sigma \setminus S$ *by:*

$$\forall x \in \Sigma \setminus S, r(x) = \min\{i \in \mathbb{O} | r \notin S_i\}$$

is a ranking function over $\Sigma \setminus S$*:*

$$\forall x \in \Sigma \setminus S, \forall y \in \Sigma, x \to y \Longrightarrow (y \notin S \wedge r(y) < r(x))$$

However, computing (or approximating) $\mathcal{S}$ may not give the successive iterates.

3.3 Abstraction and Abstract Semantics

The *template polyhedral abstraction*[27] is a parametric sub-abstraction of the classical polyhedral abstraction, where the linear constraints used must belong to a template. In practice, the template is a matrix $T \in \mathbb{R}^{m \times n}$. Each row of T represents a linear form of program variables. The matrix T defines an abstraction of $\wp(\mathbb{R}^n)$, where an abstract element ρ is a element of $\mathcal{T}_T = \overline{\mathbb{R}}^m$ where $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$.. The concretization function is defined as:

$$\gamma_T(\rho) = \{\boldsymbol{v} \in \mathbb{R}^n \mid T\boldsymbol{v} \leq \rho\}$$

Note 1. In the following, we may consider T as a matrix or as a set of m linear forms. For any $\rho \in \mathcal{T}_T$, we will denote by $[\rho]_f$ the component of ρ associated to $f \in T$.

In the context of this abstraction, the best abstract transformer $\mathrm{pre}^\sharp = \alpha_T \circ \mathrm{pre} \circ \gamma_T$ is computable:

Lemma 1. *With* $\rho \in \mathcal{T}_T$, *the component of* $\mathrm{pre}^\sharp(\rho)$ *associated to* f *satisfies* $[\mathrm{pre}^\sharp(\rho)]_f = \max_{(Q,\boldsymbol{q}) \in \mathbf{T}} [[\![(Q,\boldsymbol{q})]\!]^\sharp]_f(\rho)$ *with:*

– *If*

$$\{x | \exists x', \begin{pmatrix} Q \\ 0 \quad T \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{q} \\ \rho \end{pmatrix}\} = \emptyset,$$

then $[[\![(Q,\boldsymbol{q})]\!]^\sharp]_f = -\infty$.

– *otherwise,*

$$[[\![(Q,\boldsymbol{q})]\!]^\sharp]_f = \min\{\boldsymbol{\lambda}(\boldsymbol{q}^T \rho^T) | \boldsymbol{\lambda} \geq 0 \wedge \begin{pmatrix} Q^T & 0 \\ & T \end{pmatrix} \boldsymbol{\lambda} = \begin{pmatrix} f^T \\ 0 \end{pmatrix}\}$$

Hence $\text{pre}^\sharp(\rho)$ can be computed by solving $m.|\mathbf{T}|$ linear programs. However, the classical Kleene iterations (with widening and narrowing operators) can only be used to approximate the abstract semantics $\mathcal{S}^\sharp = \text{gfp}\,\text{pre}^\sharp$ with the following restrictions:

1. Using a *dual* widening operator can only be used to find an *underapproximation* of the abstract semantics [11], which is not sound since our abstract semantics is an *overapproximation* of the concrete semantics[1].
2. Using n steps of a narrowing operator would only find states terminating after at most n iterations.

Thus we need to use *policy iteration* to compute $\mathcal{S}^\sharp$.

4 Policy Iteration

Policy or strategy iteration [10,18,19] is a method to compute the least (or greatest) fixpoint of specific classes of monotonic operators. It can be seen as an adaptation of the classical Newton's method in the sense that the operator σ is approximated (w.r.t. a *policy selection*) by a policy σ_0 where the fixpoint of σ_0 is computable. A new policy is selected to approximate σ around the fixpoint of σ_0, and the process iterates until a global fixpoint is reached.

The application of policy iteration to the computation of greatest fixpoints has been presented in [23]. Although this section extends slightly the framework by not restricting transitions to assignments, the results are similar.

4.1 Policy Selection

Theorem 1. *Given a transition $t = (Q, \boldsymbol{q})$, $[[\![t]\!]^\sharp]_f$ is the minimum of two function ψ^t and $[\phi^t]_f$ where:*

- *ψ^t is monotonic and its image is in $\{-\infty, +\infty\}$;*
- *$[\phi^t]_f$ is the minimum of a finite number of affine expressions (with positive coefficients) on ρ.*

Example 2. For t_1 in program A, and $T = (-a, -a-b)$, we have:

$$\begin{aligned} \psi^{t_1}(\rho) &= +\infty \\ [\phi^{t_1}]_{-a}(\rho) &= -1 + \rho_{-a-b} \\ [\phi^{t_1}]_{-a-b}(\rho) &= \min(-1 + \rho_{-a-b}, \rho_{-a}) \end{aligned}$$

Theorem 2. *Let t be a transition, and $\rho \in \overline{\mathbb{R}}^m$. The value of $\psi^t(\rho)$ and, if $\psi^t(\rho) = +\infty$, the affine expression for which $\phi^t(\rho)$ is minimal can be computed by solving the following linear program:*

$$\max\{f.\boldsymbol{x} | \exists \boldsymbol{x}, \boldsymbol{x'}, \begin{pmatrix} Q \\ 0\,T \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x'} \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{q} \\ \rho \end{pmatrix}\}$$

[1] The overapproximation being necessary to prove termination.

```
$\sigma \leftarrow (+\infty)$                                  ▷ Initial policy
$\rho \leftarrow (+\infty)$                                  ▷ Initial fixpoint: $\top$
while $\Phi(\rho) \neq \rho$ do                            ▷ Stop if fixpoint is reached
    for all $f \in T$ do
        if $[\Phi(\rho)]_f < [\rho]_f$ then          ▷ Change $[\sigma]_f$ only if it is not optimal for $\rho$
            $[\sigma]_f \leftarrow$ min-policy of $[\Phi]_f$ such that $[\sigma]_f(\rho) = [\Phi]_f(\rho)$
        end if
    end for
    $\rho \leftarrow \text{gfp}_{\leq\rho}\sigma$                 ▷ Compute the next fixpoint for the policy
end while
```

Fig. 2. Policy iteration algorithm for greatest fixpoint computation. This algorithm computes gfp Φ using min-policies: each policy component $[\sigma]_f$ is given by a choice between the min-terms of $[\Phi]_f$.

If this linear program is infeasible, then $\psi^t = -\infty$, otherwise, $\psi^t = +\infty$ and ϕ^t_f can be directly constructed from the optimal dual solution.

Given a current policy σ and its fixpoint ρ, the new policy σ' is constructed as follows : $[\sigma']_f = \max_{t \in \mathbf{T}} [\sigma^t]_f$ where $[\sigma^t]_f = -\infty$ if $\psi^t(\rho_0) = -\infty$, and $[\sigma^t]_f$ is an affine expression of $[\phi^t]_f(\rho_0)$ which is minimal for ρ_0 otherwise.

Proposition 3. *The policy selection process constructs a new policy for which each component is the maximum of affine expressions.*

Example 3. For program A, if the current post-fixpoint is $(\rho_{-a} = 0, \rho_{-a-b} = 0)$, the policy selection process gives:

$$\begin{aligned}[\sigma_1]_{-a}(\rho) &= \max(\rho_{-a-b} - 1, \rho_{-a} - 1) \\ [\sigma_1]_{-a-b}(\rho) &= \max(\rho_{-a}, \rho_{-a-b} - 1)\end{aligned}$$

In the program, this policy expresses the fact that, if after an iteration the constraints $-a \leq u \leq 0$ and $-a - b \leq v \leq 0$ are satisfied, then before the iteration, the constraints $-a \leq \max(u - 1, v - 1)$ and $-a - b \leq \max(u, v - 1)$ are satisfied.

In general, we will write the policy as a system of equations over the components of ρ, e.g.:

$$\begin{aligned}\rho_{-a} &= \max(\rho_{-a-b} - 1, \rho_{-a} - 1) \\ \rho_{-a-b} &= \max(\rho_{-a}, \rho_{-a-b} - 1)\end{aligned}$$

4.2 Policy Iteration Result

Theorem 3 ([23]). *Following the algorithm of policy iteration presented Fig. 2, the new fixpoint ρ_l of each policy σ_l is computable by solving two linear programs. Furthermore, the algorithm terminates and gives the abstract fixpoint $\mathcal{S}^\sharp$.*

Note 2. Although the algorithm of Fig. 2 starts with $\sigma_0 = (+\infty)$, we can adapt it to start from any post-fixpoint.

Example 4. Figure 3 shows the results of the analyzes of programs A and B. The initial policy $((+\infty))$ is omitted. For both programs, Step 1 gives just the translation of the termination condition of the loop. Step 2 gives the equivalent of one more iteration. However, in Step 3, the relations between variables in the equation system enables to jump directly to the fixpoint. This jump is equivalent to ω iterations in the greatest fixpoint computation. Note that the analysis proves the termination of program A from every initial state. For program B, it proves that the programs terminates from $x > 1.6 \vee x < -1.2$, which is the best result we can get with a polyhedral abstraction ($x = 1.6$ and $x = -1.2$ being both non-terminating states). However, it does not give the exact set of terminating states (e.g. one can check that this program terminates from any integer).

Step	Policy	Fixpoint
1	$\rho_{-a} = 0$ $\rho_{-a-b} = +\infty$	$-a \leq 0$
2	$\rho_{-a} = \max(0, \rho_{-a} - 1)$ $\rho_{-a-b} = \rho_{-a}$	$-a \leq 0$ $-a-b \leq 0$
3	$\rho_{-a} = \max(\rho_{-a-b} - 1, \rho_{-a} - 1)$ $\rho_{-a-b} = \max(\rho_{-a}, \rho_{-a-b} - 1)$	$\emptyset$

(Prog. A)

Step	Policy	Fixpoint
1	$\rho_x = 100$ $\rho_{-r} = +\infty$	$x \leq 100$
2	$\rho_x = 100$ $\rho_{-x} = \max((\rho_x - 2)/2, (\rho_x + 2)/3))$	$x \leq 100$ $-x \leq 49$
3	$\rho_x = \max((\rho_{-x} + 2)/2, (\rho_{-x} - 2)/3)$ $\rho_{-x} = \max((\rho_x - 2)/2, (\rho_x + 2)/3))$	$x \leq 1.6$ $-x \leq 1.2$

(Prog. B)

Fig. 3. Results of the policy iteration process on program A with the template $T = (-a, -a-b)$ and program B with the template $T = (-x, x)$

5 Relationships with Ranking Functions

The policy iteration process computes the exact abstract semantics of the program. Proposition 2 states that this fixpoint entails the existence of a ranking function based on the iterates of the fixpoint, but does not give the form of the ranking function (or relation). In order to compare this approach with other existing methods, we need to make this ranking function explicit, or at least precise the kind of ranking relations a program must satisfy to be successfully analyzable by policy iteration. Since we use linear templates, linear ranking functions and their derivatives (piecewise linear ranking functions and disjunctive linear ranking relations) are the most interesting candidates to compare with our approach.

5.1 Ranking Functions and Relations

Linear Ranking Function. Linear ranking functions are commonly used to prove termination of simple programs.

Definition 1 (Ranking function). *If Σ is a set of states, S a subset of Σ, and $\tau \subseteq \Sigma \times \Sigma$ a transition relation, a* ranking function *over S is defined by an ordered set $(O, \prec)$ and a function $r : S \to O$ such that $\prec$ is a well-founded order and:*

$$\forall \sigma \in S, \sigma \xrightarrow{\tau} \sigma' \;\Rightarrow\; \sigma' \in S \text{ and } r(\sigma') \prec r(\sigma)$$

If τ represents the transition relation induced by a program P, the existence of a ranking function over S shows that the program terminates from any state in S.

The *ranking relation* $T(r) \subseteq S \times S$ generated by a ranking function r is the well-founded relation defined as $T(r) = \{(\sigma_1, \sigma_2) \mid r(\sigma_2) \prec r(\sigma_1)\}$. If r is a ranking function for τ over S, then $T(r)$ satisfies:

$$\tau \;\subseteq\; T(r) \cup (\Sigma \setminus S) \times \Sigma$$

We may use ordinals $(\mathbb{O}, <)$ as well-founded sets. However, since our approach deals with real values, we will denote by $\prec$ on a subset of $\mathbb{R}$, any well-founded suborder of $<$ on this subset.

Definition 2 (Linear ranking function). *A ranking function r on $(O, \prec)$ is* linear *if O is a subset of $\mathbb{R}$, $\prec$ is a sub-order of $<$ on O, and r is linear.*

Segmented Linear Ranking Functions. The domain of segmented ranking functions is presented in [30] to infer termination properties on programs. Its analysis produces piecewise-segmented ranking functions to infer sufficient conditions on programs. The domain is parametrized by two numerical abstract domains for the partitioning of the environment and for the values of the function. The prototypes used intervals for the partitioning and affine forms for the functions, but it should be possible to use other linear constraints (maybe templates) for the partitioning (which should be costly). We propose to call this instantiation of the domain *segmented linear ranking functions.*

Definition 3 (Segmented linear ranking function). *Let $S \subseteq \mathbb{R}^m$ and τ a transition relation on $\mathbb{R}^m$, a ranking function $r : S \to O$ on S is segmented linear if it can be defined by a n-uplet $(S_1, r_1), \ldots, (S_n, r_n)$ where:*

- *$\{S_i\}_{1 \leq i \leq n}$ is a partition of S, and each S_i is a polyhedron;*
- *for all i, r_i is defined on S_i and $r = r_i$ on S_i.*
- *all the r_i are linear.*

Disjunctive Linear Ranking Relations. An alternative to ranking functions are disjunctive ranking relations, defined as a finite union of ranking relations $T = T_1 \cup \ldots \cup T_n$. Although T may not be itself a ranking relation, a transition relation τ is well-founded if and only if its non-reflective transitive closure τ^+ is included in a disjunctive ranking relation (which is then called a *transition invariant*[25]). This approach is widely used to prove termination, using model-checking procedures to check the inclusion in transition invariants [22,8]. Of course, disjunctive ranking functions can be used to prove conditional termination:

Lemma 2. *Let P a program with a transition relation $\tau \subseteq \Sigma \times \Sigma$. Then P terminates from all states in S if and only if there exists a disjunctive ranking relation $T = T_1 \cup \ldots \cup T_n$ such that:*

$$\tau^+ \subseteq T \cup (\Sigma \setminus S) \times \Sigma$$

In [5], Chen *et al.* proposed to infer disjunctive ranking relations for linear simple loops (LSLs), where each ranking relation T_i is (by construction) based on a linear ranking function. More precisely, $T_i = T(r_i)$ such that there exists a polyhedral partition $(P_1, \ldots, P_k)$ of $\mathbb{R}^n$ where:

– r_i is linear over P_1;
– r_i is constant over P_2, ..., P_k and its values are always strictly lower than the elements of $r_i(P_1)$.

We will call these relations *disjunctive linear ranking relations.*

Before considering the ranking functions induced by the policy iteration algorithm, we examine the relationships between disjunctive ranking relations and segmented ranking functions. Disjunctive linear ranking relations are strictly more powerful than segmented linear ranking functions: any segmented linear ranking function induces a disjunctive linear ranking relation, but the converse is not true.

Theorem 4. *Let $\tau \subseteq \Sigma \times \Sigma$ and r a segmented linear ranking function for τ over S, defined by the n-uplet (S_1, r_1), ..., (S_n, r_n). For all $1 \leq i \leq n$, we define $T_i : S \times S$ as:*

$$T_i = T(r_i) \cup (S_i \times (S \setminus S_i))$$

Then $\tau^+ \subseteq T_1 \cup \ldots \cup T_n \cup (\Sigma \setminus S) \times \Sigma$.

Example 5. Program A admits a segmented linear ranking function r defined as:

$$r(a,b) = \begin{cases} 0 & \text{if } a < 0 \\ 1 & \text{if } a \geq 0 \text{ and } a + b < 0 \\ 2a + 2 & \text{if } a \geq 0 \text{ and } b \geq 0 \\ 2(a+b) + 3 & \text{if } a + b \geq 0 \text{ and } b < 0 \end{cases}$$

The partition contains four sets. The disjunctive ranking relation allows any transition between two different sets, but only decreasing transitions (w.r.t. the local ranking function) inside one set. While not well-founded, it is the union of well-founded relations and includes the transition closure of τ.

The disjunctive ranking relation does not give any information about the relations between the elements of the partition, therefore it can prove the termination of programs which do not admit a segmented linear ranking function.

Example 6. The program

```
1: while $x \geq 0$ do
2:     $x \leftarrow x + y$
3:     if $y \geq 0$ then
4:         $y \leftarrow y - 1$
5:     end if
6: end while
```

terminates and admits a disjunctive ranking relation defined by the ranking functions:

$$\rho_0 = \begin{cases} x & \text{if } y \leq -1 \text{ and } x \geq 0 \\ -1 & \text{if } x < 0, \text{ or } y > 0 \end{cases} \qquad \rho_1 = \begin{cases} y \text{ if } y > 0 \\ 0 \text{ if } y \leq 0 \end{cases}$$

However, a segmented ranking function would need to be quadratic when $y > 0$.

5.2 Policy Iteration and Ranking Relations

Our goal is to link the results of the policy iteration analysis with the existence of disjunctive ranking relations or segmented ranking functions. Since the policy iteration analysis gives an overapproximation of the non-terminating states, there exists a ranking function or relation on the complement, which should be related to the template used.

Example 7. For program B, the policy iteration analysis (with $T = (x, -x)$) proves that the program terminates from $x > 1.6$ or $x < -1.2$. A ranking function r should be defined on $]-\infty, -1.2[\cup]1.6, +\infty[$. Since $|x|$ increases at each iteration, r should be increasing on $]-\infty, -1.2[$ (with $\lim_{x \to -1.2} r(x) = \omega$) and decreasing on $]1.6, +\infty[$ (with $\lim_{x \to 1.6} r(x) = \omega$). Hence, on $]-\infty, -1.2[$, x is a ranking function, whereas $-x$ is a ranking function on $]1.6, +\infty[$. Note that the value -1.2 is given by ρ_{-x} and 1.6 by ρ_x. Therefore, it appears that the partial ranking functions are related to the negation of the template elements.

In general, we shall prove that if the policy iteration analysis shows that A is a set of terminating states:

1. A disjunctive linear ranking relation can be defined on A, of which the ranking relations are directly related to the template (Theorem 5).
2. We can also find on A a segmented linear ranking function r, with template-related restrictions of the domains of the subfunctions. To represent these restrictions, we shall describe r as a min-defined segmented ranking function, i.e. as a minimum of functions with overlapping domains (Definition 4). Furthermore, we prove the converse of this result, i.e. the existence of a

segmented linear ranking function satisfying these restrictions on A implies that the policy iteration analysis can prove conditional termination on A (Theorem 6).

By Theorem 3, we know that the policy iteration process returns the exact abstract semantics of the program, defined as $\mathcal{S}^\sharp = \text{gfp } \alpha_T \circ \text{pre} \circ \gamma_T$. The iterates of this fixpoint are elements of the template abstract domain. Hence we can expect a potential ranking relation to be closely related to the template linear forms. Theorem 5 formalizes this idea and shows that the programs directly admits a disjunctive linear ranking relation on the terminating part:

Theorem 5. *Let T be a template with m linear forms $f_1, \ldots, f_m$ over $\mathbb{R}^{\boldsymbol{x}}$, and $A^\sharp$ an abstract element of $\mathcal{T}_T$. If $\text{gfp}\, \lambda X.(\text{pre}_T^\sharp(X)) = A^\sharp$, then there exists m ranking relations R_1, R_2, ..., R_m on $\mathbb{R}^{\boldsymbol{x}} \setminus \gamma_T(A^\sharp)$ satisfying the conditions (C1) and (C2) defined as follows:*

(C1) *For all i, there exists a well-founded suborder $\prec^i$ of $<$ on $\mathbb{R}$ such that:*

$$\forall(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \mathbb{R}^n \setminus \gamma_T(A^\sharp), (\boldsymbol{v}_1, \boldsymbol{v}_2) \in R_i \iff -f_i\boldsymbol{v}_2 \prec^i -f_i\boldsymbol{v}_1$$

(C2) *$R_1 \cup \ldots \cup R_m$ is a disjunctive ranking relation for τ over $\mathbb{R}^n \setminus \gamma_T(A^\sharp)$:*

$$\tau^+ \subseteq (R_1 \cup \ldots \cup R_m) \cup \gamma_T(A^\sharp) \times \mathbb{R}^n$$

Furthermore, with $(\text{pre}_T^\sharp)^k$ denoting the k-th iterate of the operator $\text{pre}_T^\sharp$ starting from $\mathbb{R}^n$, we can construct $\prec^i$ as:

$$u \prec^i v \iff \exists k \in \mathbb{O}, u < -[(\text{pre}_T^\sharp)^k]_{f_i} \leq v$$

This theorem proposes well-founded orders $\prec^i$ based on sets of iterates, which are not easy to use. We shall examine the problem of finding simpler orders in Sect. 5.3.

Example 8. We can prove the termination of program A with the template $T = (-a, -a-b)$. The associated relations R_1 and R_2 can be defined on $\mathbb{R}^2$ as follows:

$$((a,b),(a',b')) \in R_1 \text{ iff } a' \prec a$$
$$((a,b),(a',b')) \in R_2 \text{ iff } a' + b' \prec a + b$$

where

$$u \prec u' \iff a < 0 \leq b \vee 0 \leq a + 1 \leq b$$

We can see that R_1 and R_2 are generated by the functions a and $a + b$.

Example 9 shows that the converse of Theorem 5 does not hold:

Example 9. The program seen in Example 6 admits a disjunctive ranking relation

$$T(\rho_0) \cup T(\rho_1) \text{ with } \rho_0 = x \text{ and } \rho_1 = y$$

with the well-founded order $\prec$ defined as $a \prec b \Leftrightarrow a < 0 \leq b \vee a + 1 \leq b$. However, analyzing the program with the template $T = (-x, -y)$ gives just the condition $x \geq 0$ for potential non-termination, because the next iterate in the concrete domain gives the constraint $x + y \geq 0$ which is not translated in the abstract domain.

Theorem 5 presents a disjunctive linear ranking relation of which the components are based on the policy iteration analysis. As we saw on Example 6, this does not prove the existence of a segmented linear ranking function. Two difficulties arise when we try to construct a segmented ranking function from the policy iteration analysis.

First, the domains of the subfunctions must partition the terminating states, whereas each well-founded order of Theorem 5 is defined on the whole domain. We circumvent the problem by including the possibility of overlapping domains. In this case, the value of the r is defined as the minimum of the values of the underlying functions.

Definition 4 (Min-defined segmented ranking functions). *Let Σ be a set of states, S a subset of Σ and τ a transition relation on Σ, a ranking function $r : S \to O$ on S (where $O \subseteq \mathbb{R}$) is min-defined segmented if it can be defined by a n-uplet $(S_1, r_1), \ldots, (S_n, r_n)$ where:*

1. $\bigcup_{1 \leq i \leq n} S_i = S$,
2. *and $\forall \sigma \in S, r(\sigma) = \min_{\sigma \in S_i} r_i(\sigma)$ (where* min *is the minimum with respect to the total order $<$).*

Furthermore, r is min-defined segmented linear if all r_i are linear.

Of course, any min-defined segmented ranking function can be transformed to a segmented ranking function by restricting the domain of the subfunctions. However, using them makes the following theorem much easier to present, as the domain of each subfunction becomes independent of the others subfunctions.

The second difficulty is the relationships between the values of different subfunctions. The easiest approach is to consider intermediate functions φ:

Theorem 6. *Let $A^\sharp \in \mathcal{T}_T$. Then* $\text{gfp}\,\lambda X.(\text{pre}^\sharp_T(X)) \sqsubseteq^\sharp A^\sharp$ if and only if *there exists a min-defined segmented ranking function $r = \min_{i \in \{1,\ldots,m\}} r_i : \mathbb{R}^n \setminus \gamma_T(A^\sharp) \to O$ such that for all i:*

(C3) $\text{Dom}(r_i) = \{\boldsymbol{v} \in \mathbb{R}^n \mid [A^\sharp]_{f_i} < f_i\boldsymbol{v}\}$
(C4) $\forall i, r_i(\boldsymbol{v}) = \varphi_i(-f_i\boldsymbol{v})$ *where φ_i is a monotonic function from $]-\infty, -[A^\sharp]_{f_i}[$ to O.*

Proof (sketch). Let's consider the iterates $(A^\sharp_k)$ of $pre^\sharp_T$ starting from $(+\infty)$. We define $r_i(\boldsymbol{v})$ as the maximal ordinal k such that $f_i\boldsymbol{v} \leq [A^\sharp_k]_{f_i}$. One can easily check that this ordinal exists (if $\boldsymbol{v} \in \text{Dom}(r_i)$), that r_i is monotonic w.r.t. $-f_i\boldsymbol{v}$, and that r is a ranking function. Reciprocally, if a ranking function r satisfy (C3) and (C4), then we prove by transfinite induction that for all $\boldsymbol{v} \in \gamma_T(A^\sharp_k)$, $r(\boldsymbol{v}) \geq k$. The limit case is a consequence of the co-continuity of γ_T. For the successor case, let us suppose that there exists $\boldsymbol{v} \in \gamma_T(A^\sharp_{k+1}) = \gamma_T \circ \alpha_T(\text{pre}(\gamma_T(A^\sharp_k)))$ such that $r(\boldsymbol{v}) < k+1$ (for example, $r_i(\boldsymbol{v}) \leq k$). Then there exists $\boldsymbol{v}' \in \text{pre}(\gamma_T(A^\sharp_k))$ such that $f_i\boldsymbol{v}' \geq f_i\boldsymbol{v}$, hence $r_i(\boldsymbol{v}')$ is defined by (C3) and $r(\boldsymbol{v}') \leq r_i(\boldsymbol{v}') \leq r_i(\boldsymbol{v}) \leq k$ by (C4). Since $\boldsymbol{v}' \in \text{pre}(\gamma_T(A^\sharp_k))$, there must be a successor $\boldsymbol{v}''$ of $\boldsymbol{v}'$ in $\gamma_T(A^\sharp_k))$, which by induction hypothesis must satisfy $r(\boldsymbol{v}'') \geq k$, which contradicts the fact that r is a ranking function.

Example 10. Example 5 gives a segmented ranking r function for program A, which can also be min-defined:

$$r(a,b) = \min(r_{-a}(a,b), r_{-a-b}(a,b))$$

with r_{-a} (resp. r_{-a-b}) depending only on a (resp. $a+b$):

$$r_{-a}(a,b) = \begin{cases} 0 & \text{if } a < 0 \\ 2a+2 & \text{if } a \geq 0 \end{cases}$$

$$r_{-a-b}(a,b) = \begin{cases} 1 & \text{if } a+b < 0 \\ 2(a+b)+3 & \text{if } a+b \geq 0 \end{cases}$$

Theorem 6 shows the form of a potential ranking function for the set of terminating states found by the policy iteration algorithm. Also, it gives a completeness result by describing the programs analyzable with a specific template as any program admitting a min-defined segmented ranking function satisfying these conditions. Since linear ranking functions satisfy them, we can deduce that programs admitting linear ranking functions can be proved to terminate by policy iteration (with an appropriate template):

Corollary 1. *If a program admits a linear ranking function r, then a policy iteration analysis with a template T including $-r$ can prove its termination.*

Conditional termination with a linear ranking function, however, is more complicated since condition (C3) make strong assumptions on the domain of the ranking function.

Example 11. The program

```
1: while $x \geq 0$ do
2:     $x \leftarrow x + y$
3: end while
```

admits x as a linear ranking function when $y < -1$. However, an analysis with the template $(-x, -y)$ cannot give any result of the form $A^\sharp = (x >= 0, y >= -1)$ because the domain of the ranking function associated to x should be all the states satisfying $x < 0$. More generally, while the set of non-terminating states is $(x >= 0, y >= 0)$, we can prove that no template can give this result.

On the other hand, with an initial constraint of the form $y \leq -\epsilon < 0$, the analysis proves the termination of the program.

5.3 On the Well-founded Relations

Theorem 5 proposes well-founded relations $\prec^i$ defined from (infinite) sets of iterates of the abstract computation. However, relations found in the literature are generally defined directly. For example, a common order used in linear ranking functions is $\prec_\epsilon$ (with $\epsilon > 0$) defined as:

$$a \prec_\epsilon b \iff a + \epsilon \leq b$$

Such an order is useful because checking $a \prec_\epsilon b$ is easier, and because it shows the evolution of the states towards termination.

In this section we study the possibility of constructing theses kinds of orders (not based on infinite sets) from the analysis. Our first step is consider the sequence of policies. This sequence constructs a finite and decreasing chain of $p+1$ fixpoints $\rho_0 = (+\infty), \ldots, \rho_p$. Projecting this chain to the i-th component gives a decreasing sequence $\rho_0^i = +\infty, \ldots, \rho_p^i$. As a result, any couple (a, b) where $a < -\rho_k^i \leq b$ can be included in $\prec^i$. Hence we can construct $\prec^i$ as:

$$\begin{aligned} \prec^i = \prec^{i,1} \ \cup \ \prec^{i,2} \ \cup \ \ldots \ \cup \ \prec^{i,p} \\ \cup \ ]-\infty, -\rho_1^i[\times [-\rho_1^i, +\infty[\ \cup \ \ldots \ \cup \ ]-\infty, -\rho_p^i[\times[-\rho_p^i, +\infty[\end{aligned} \tag{1}$$

where each $\prec^{i,k}$ is associated to the k-th policy and only defined on $[-\rho_{k-1}^i, -\rho_k^i[$.

Example 12. With program B, we need two well-founded orders $\prec^x$ and $\prec^{-x}$ on $\mathbb{R}$. Since the successive fixpoints (for the linear form x) gives $x \leq 100$ and $x \leq 1.6$ (cf. Fig. 3), we may construct $\prec^x$ as:

$$\begin{aligned} a \prec^x b \Leftrightarrow \ & a < -1.6 \leq b \\ & \text{or } a < -100 \leq b \\ & \text{or } (a, b) \in [-100, -1.6[\ \wedge \ a \prec^{x,3} b \\ & \text{or } (a, b) \in]-\infty, -100[\ \wedge \ a \prec^{x,1} b \end{aligned}$$

where $\prec^{x,1}$ (resp. $\prec^{x,3}$) is a well-founded suborder of $<$ on $]-\infty, -100[$ (resp. $[-100, -1.6[$).

Finding an order for each policy is difficult. Let's restrict ourselves to the case of a LSL. The policy is described as a system of affine equations, and the next fixpoint as the limit of a sequence $(\boldsymbol{\nu}_k)$ of the form:

$$\begin{aligned} \boldsymbol{\nu}_0 &= \rho_j \\ \boldsymbol{\nu}_{k+1} &= A\boldsymbol{\nu}_k + B \end{aligned}$$

where A is a nonnegative matrix. Then we have $\boldsymbol{\nu}_{k-1} - \boldsymbol{\nu}_k = A^k(\boldsymbol{\nu}_0 - \boldsymbol{\nu}_1)$ (where $\boldsymbol{\nu}_0 - \boldsymbol{\nu}_1$ is also nonnegative). Using [20, Sect. 9.3], we get the following proposition:

Proposition 4. *Let P be a LSL, and ρ_0, ..., ρ_p the sequence constructed by policy iteration on P, ρ_{k-1}^i and ρ_k^i the i-th component of ρ_{k-1} and ρ_k. Then there exists a well-founded order $\prec^{i,k}$ for equation (1) and a integer $d > 0$ such that:*

– *if* $\rho_k = -\infty$,

$$\exists \epsilon > 0, (\prec^{i,k})^d \subseteq \prec_\epsilon \quad \textit{where } a \prec_\epsilon b \Leftrightarrow a + \epsilon \leq b$$

– *otherwise:*

$$\exists h > 1, (\prec^{i,k})^d \subseteq \prec_{h,-\rho^i_k} \quad \textit{where } a \prec_{h,-\rho^i_k} b \Leftrightarrow (-\rho^i_k - a) \geq h(-\rho^i_k - b)$$

In this proposition, $(\prec^{i,k})^d = \prec^{i,k} \circ \ldots \circ \prec^{i,k}$ is the d-th power of $\prec^{i,k}$. Note that $\prec_\epsilon$ or $\prec_{h,-\rho^i_k}$ are well-founded orders on $[-\rho^i_{k-1}, -\rho^i_k[$, which implies that $\prec^{i,k}$ is a well-founded order.

This proposition does not directly give $\prec^{i,k}$, but it shows that decreasing sequences decreases (at least) on average linearly when $\rho_k = -\infty$ and geometrically (from the upper bound) when ρ_k is finite. Although this results is not proven on the general case, we expect the progressions to be similar in most cases.

Example 13. Continuing the previous example, we consider $\prec^{x,1}$ on $]-\infty, -100[$ and $\prec^{x,3}$ on $[-100, -1.6[$. The first policy (associated to $\prec^{x,1}$) returns the next fixpoint after one iteration. Hence we can just define $\prec^{x,1} = \emptyset$.

The third policy (associated to $\prec^{x,3}$) converges with a geometric rate. The order $\prec^{x,3}$ can be defined as:

$$u \prec^{x,3} u' \iff (-1.6 - u) \geq 4(-1.6 - u')$$

Concretely, this results shows that from an initial value $x > 1.6$, the value of x (when $x > 0$) diverges from 1.6 at a geometric rate.

6 Experiments

6.1 Template Selection

A prototype analyzer implementing the policy iteration algorithm was developed, using VPL[2] to handle exact polyhedral and linear programming operations. Since our analysis use the template polyhedral domain, selecting a correct template was an issue. However, overapproximating the greatest fixpoint enables a progressive refinement of the template: any fixpoint computed with a template can be used as a starting post-fixpoint with another template. Based on this idea, the following heuristics was implemented:

1. first, a few backwards steps in the general polyhedral domain is computed;
2. then the actual linear constraints are used as a basis for the template and the abstract fixpoint is computed for this template;
3. the process can be iterated from the new post-fixpoint, alternating between backward direct iterations and policy iterations. Since every intermediate result is a safe approximation, we can stop anytime, or if the fixpoint is reached.

[2] `http://verasco.imag.fr/wiki/VPL`

This heuristics can be compared to existing techniques for invariant analyses which uses partial traces to specialize the abstract domain [3,28]. In the worst case, the analysis gives the same result as a classical narrowing, returning an approximation of the states which do not terminate after n loop iterations. In the best case, it gives the abstract greatest fixpoint in the (general) polyhedral domain, which may not be the exact set of non-terminating states.

Example 14. In the terminating program (from [9]):

```
while $x \neq 0$ do
    if $x > 0$ then
        $x \leftarrow x - 1$
    else
        $x \leftarrow x + 1$
    end if
end while
```

the decreasing iterations in the polyhedral domain stabilizes immediately at $]-\infty, +\infty[$. The problem can be solved by using several program points (or, similarly, state partitioning) to separate the cases $x > 0$ and $x < 0$.

6.2 Results

LSL Test Suite. First, we used the LSL test suite proposed by Chen *et al.*[5]. This suite has 38 LSL loops, of which 12 are non-terminating, 7 are terminating with linear ranking functions, and 19 are terminating with non-linear ranking functions. Our prototype analyzed the whole test suite in 0.5 second. The results are summarized on Table 1. Terminating LSLs with a linear ranking function are all proved to terminate (although our heuristics does not guarantee this). Terminating LSLs without a linear ranking function are proved to terminate most of the time, yet our analyzer failed in 6 cases whereas Chen *et al.*'s algorithm, which is specifically designed to prove termination on LSLs, failed only twice. Interpreting the results of non-terminating LSLs is harder since this test suite was not designed for conditional termination analysis. For 2 LSLs, our approach managed to find the exact set of terminating states, something which was not possible with only narrowings. For 3 LSLs, the greatest fixpoint is directly reached by a few iterations in the polyhedral domain. Finally, for the other programs, the PI techniques does not refine the decreasing iteration sequence.

Table 1. Experiment results

LSL test suite[5]			
Programs			Results
38 LSLs	Terminating : 26	Linear r.f.: 7	Termination proved: 7
		Non-linear r.f.: 19	Termination proved: 13
	Non-terminating : 12		Exact semantics with PI: 2
			Exact semantics with narrowing: 3
			No improvement: 7

These results show that our approach is quite fast and can find complex termination properties, but not as efficient as a technique specifically designed for linear simple loops.

Other Programs. To our knowledge, no test suite exists for termination analysis (or conditional termination) on general programs. Hence we tested some examples given on previous works. The analyzes were fast and sometimes successful. However, several cases failed to give interesting results. We identified two main causes.

1. The iterates in the polyhedral domain stabilize after a few iterations, as in Example 14. This problem is directly related to the use of polyhedral abstractions.
2. Termination (or interesting conditional termination) requires the use of lexicographic ordering [9]. Our approach seems to be more suited to prove termination when the ranking relations are interlinked than when they form a lexicographic order. This is especially interesting as other approaches (limited to termination analysis) are specifically designed for lexicographic ordering [9,2]. Hence our approach can be used in complement to those.

For both problems, partitioning the set of states should improve the results.

7 Conclusion

This paper has described how policy iteration can be used to find conditional termination properties. The analysis is fast and the result can be precise, although it relies heavily on the abstract domain used. To improve the analysis, we plan to investigate the application of dynamic trace partitioning [26] for conditional termination. Another possibility would be to extend the policy iteration framework to other abstract domains such as the generalized template domain [6] or quadratic templates [1].

Acknowledgements. The author thanks D. Monniaux, L. Gonnord and S. Putot as well as the anonymous referees for their comments and suggestions.

References

1. 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)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, Martel (eds.) [15], pp. 117–133
3. Amato, G., Parton, M., Scozzari, F.: Deriving numerical abstract domains via principal component analysis. In: Cousot, Martel (eds.) [15], pp. 134–150

4. Bozga, M., Iosif, R., Konečný, F.: Deciding conditional termination. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 252–266. Springer, Heidelberg (2012)
5. Chen, H.Y., Flur, S., Mukhopadhyay, S.: Termination proofs for linear simple loops. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 422–438. Springer, Heidelberg (2012)
6. Colón, M.A., Sankaranarayanan, S.: Generalizing the template polyhedral domain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 176–195. Springer, Heidelberg (2011)
7. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
8. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
9. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 47–61. Springer, Heidelberg (2013)
10. 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)
11. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France (March 21, 1978)
12. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: Conference Record of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Philadelphia, PA, January 25-27, pp. 245–258. ACM Press, New York (2012)
13. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot (ed.) [14], pp. 1–24
14. Cousot, R. (ed.): VMCAI 2005. LNCS, vol. 3385. Springer, Heidelberg (2005)
15. Cousot, R., Martel, M. (eds.): SAS 2010. LNCS, vol. 6337. Springer, Heidelberg (2010)
16. Dai, L., Xia, B.: Non-termination sets of simple linear loops. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 61–73. Springer, Heidelberg (2012)
17. De Nicola, R. (ed.): ESOP 2007. LNCS, vol. 4421. Springer, Heidelberg (2007)
18. Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static analysis by policy iteration on relational domains. In: De Nicola (ed.) [17], pp. 237–252
19. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: De Nicola (ed.) [17], pp. 300–315
20. Hogben, L.: Handbook of Linear Algebra, 1st edn. Discrete Mathematics and Its Applications. Chapman & Hall/CRC (2007)
21. 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)
22. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)

23. Massé, D.: Proving termination by policy iteration. Electr. Notes Theor. Comput. Sci. 287, 77–88 (2012)
24. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
25. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society (2004)
26. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5) (2007)
27. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot (ed.) [14], pp. 25–41
28. Seladji, Y., Bouissou, O.: Fixpoint computation in the polyhedra abstract domain using convex and numerical analysis tools. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 149–168. Springer, Heidelberg (2013)
29. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004)
30. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 43–62. Springer, Heidelberg (2013)

Widening for Control-Flow

Ben Hardekopf[1], Ben Wiedermann[2], Berkeley Churchill[3], and Vineeth Kashyap[1]

[1] University of California, Santa Barbara
{benh,vineeth}@cs.ucsb.edu
[2] Harvey Mudd College
benw@cs.hmc.edu
[3] Stanford University
bchurchill@cs.stanford.edu

Abstract. We present a parameterized widening operator that determines the control-flow sensitivity of an analysis, i.e., its flow-sensitivity, context-sensitivity, and path-sensitivity. By instantiating the operator's parameter in different ways, the analysis can be tuned to arbitrary sensitivities without changing the abstract semantics of the analysis itself. Similarly, the analysis can be implemented so that its sensitivity can be tuned without changing the analysis implementation. Thus, the sensitivity is an independent concern, allowing the analysis designer to design and implement the analysis without worrying about its sensitivity and then easily experiment with different sensitivities after the fact. Additionally, we show that the space of control-flow sensitivities induced by this widening operator forms a lattice. The lattice meet and join operators are the product and sum of sensitivities, respectively. They can be used to automatically create new sensitivities from existing ones without manual effort. The sum operation in particular is a novel construction, which creates a new sensitivity less precise than either of its operands but containing elements of both.

1 Introduction

A program analysis designer must balance three opposing characteristics: soundness, precision, and tractability. An important dimension of this tradeoff is *control-flow sensitivity*: how precisely the analysis adheres to realizable program execution paths. Examples include various types of *path sensitivity* (e.g., property simulation [9] and predicate abstraction [2]), *flow sensitivity* (e.g., flow-insensitive [3] and flow-sensitive [13]), and *context sensitivity* (e.g., *k*-CFA [26] and object sensitivity [20]). By tracking realizable execution paths more precisely, the analysis may compute more precise results but also may become less tractable. Thus, choosing the right control-flow sensitivity for a particular analysis is crucial for finding the sweet-spot that combines useful results with tractable performance.

We present a set of insights and formalisms that allow control-flow sensitivity to be treated as an independent concern, separately from the rest of the analysis design and implementation. This separation of concerns allows the analysis designer to empirically experiment with many different analysis sensitivities in a guaranteed sound manner, without modifying the analysis design or implementation. These sensitivities are not restricted to currently known strategies; the designer can easily develop and experiment with new sensitivities as well. Besides allowing manual exploration of potential new sensitivities, we also describe a mechanism to automatically create new sensitivities, based on the insight that the space of control-flow sensitivities forms a lattice. The meet and join operators of this lattice can be used to construct novel sensitivities from existing ones without requiring manual intervention.

Key Insights. Our key insight is that control-flow sensitivity is a form of widening, and that we can exploit this to separate control-flow sensitivity from the rest of the analysis. This paper describes control-flow sensitivity as a widening operator parameterized by an equivalence relation that partitions states according to an abstraction of the program's history of computation. This widening-based view of control-flow sensitivity has both theoretical and practical implications: it generalizes and modularizes existing insights into control-flow sensitivity, and provides the analysis designer with a method for implementing and evaluating many possible sensitivities in a modular way.

A common technique to formalize control-flow sensitivity is to abstract a program's concrete control flow as an abstract trace (i.e., some notion of the history of computation that led to a particular program point). There are many ways to design such an abstraction, including ad-hoc values that represent control-flow (e.g., the timestamps of van Horn and Might [28]), designed abstractions with a direct connection to the concrete semantics (e.g., the mementoes of Nielson and Nielson [21]), and calculated abstractions that result from the composition of Galois connections (e.g., the 0-CFA analysis derived by Midtgaard and Jensen [18]). Existing formalisms are also tied to the notion of abstraction by partitioning [8]: the control-flow abstraction partitions the set of states into equivalence relations, the abstract values of which are merged.

Our formalisms follow this general approach (tracing and partitioning). However, prior work starts from a subset of known control-flow approximations (e.g, context-sensitivity [14,21,27], 0-CFA [18], or various forms of k-limiting and store value-based approximations [15,23]) and seeks to formalize and prove sound those specific control-flow approximations for a given analysis. In addition, most prior work calculates a series of Galois connections that leads to a specific (family of) control-flow sensitivity. In contrast, our work provides a more general view that specifies a superset of the control-flow sensitivities specified by prior work and exposes the possibility of many new control-flow sensitivities, while simplifying the required formalisms and enabling a practical implementation based directly on our formalisms. As such, our work has similar goals to Might and Manolios' *a posteriori* approach to soundness, which separates many aspects of the precision of an analysis from its soundness [19]; however, our technique relies on a novel insight that connects widening and control-flow sensitivity.

Contributions. This paper makes the following specific contributions:

- A new formulation of control-flow sensitivity as a widening operator, which generalizes and modularizes existing formulations based on abstraction by partitioning. This formulation leads to a method for designing and implementing a program analysis so that control-flow sensitivity is a separate and independent component. The paper describes several requirements on the form a semantics should take to enable separable control-flow sensitivity. Individually these observations are not novel;

in fact, they may be well-known to the community. When collectively combined, however, they form an analysis design that permits sound, tunable control-flow approximation via widening. (Section 2)
- A novel way to automatically derive new control-flow sensitivities by combining existing ones. Our results follow from category theoretic constructions and reveal that the space of control-flow sensitivities forms a lattice. (Section 3)
- An in-depth example that applies our method to a language with mutable state and higher-order functions, creating a tractable abstract interpreter with separate and tunable control-flow sensitivity. We describe several example trace abstractions that induce well-known sensitivities. We also illustrate our example with an accompanying implementation, available in the supplemental materials.[1] (Section 4)

2 Separating Control-Flow Sensitivity from an Analysis

In this section, we describe how to use widening to separate control-flow sensitivity from the rest of the analysis and make it an independent concern. We first establish our starting point: an abstract semantics that defines an analysis with no notion of sensitivity. We then describe a parameterized widening operator for the analysis and show how different instantiations of the parameter yield different control-flow sensitivities. Finally, we discuss some requirements on the form of semantics used by the analysis that make it amenable to describing control-flow sensitivity. The discussion in this section leaves the exact language and semantics being analyzed unspecified; Section 4 provides a detailed concrete example of these concepts for a specific language and semantics.

2.1 Starting Point

This subsection provides background and context on program analysis, giving us a starting point for our design. Nothing in this subsection is novel, the material is adapted from existing work [6]. For concreteness, we assume that the abstract semantics is described as a state transition system, e.g., a small-step abstract machine semantics; Section 2.4 will discuss more general requirements on the form of the semantics. The abstract semantics is formally described as a set of states $\hat{\varsigma} \in \Sigma^\sharp$ and a transition relation between states $\mathcal{F}^\sharp \subseteq \Sigma^\sharp \times \Sigma^\sharp$. The semantics uses a transition relation instead of a function to account for nondeterminism in the analysis due to uncertain control-flow (e.g., when a conditional guard's truth value is indeterminate, and so the analysis must take both branches). The set of states forms a lattice $\mathcal{L}^\sharp = (\Sigma^\sharp, \sqsubseteq, \sqcap, \sqcup)$. We leave the definition of states and the transition relation unspecified, but we assume that any abstract domains used in the states are equipped with a widening operator.[2]

The program analysis is defined as the set of all reachable states starting from some set of initial states and iteratively applying the transition relation. This definition is formalized as a least fixpoint computation. Let $\mathring{\mathcal{F}}^\sharp(S) \overset{\mathsf{def}}{=} S \cup \mathcal{F}^\sharp(S)$, i.e., a relation that is lifted to remember every state visited by the transition relation $\mathcal{F}^\sharp$. The analysis of a

[1] `http://www.cs.ucsb.edu/~pllab`, under the `Downloads` link.

[2] If the domain is a noetherian lattice then the lattice join operator is a widening operator.

program P is defined as $[\![P]\!]^\sharp \stackrel{\mathsf{def}}{=} \mathbf{lfp}_{\Sigma_I^\sharp} \mathring{\mathcal{F}}^\sharp$, i.e., the least fixpoint of $\mathring{\mathcal{F}}^\sharp$ starting from an initial set of states $\Sigma_I^\sharp$ derived from P.

The analysis $[\![P]\!]^\sharp$ is intractable, because the set of reachable states is either infinite or, at the least, exponential in the number of nondeterministic transitions made during the fixpoint computation. The issue is control-flow—specifically, the nondeterministic choices that must be made by the analysis: which branch of a conditional should be taken, whether a loop should be entered or exited, which (indirect) function should be called, etc. The analysis designer at this point must either (1) bake into the abstract semantics a specific strategy for dealing with control-flow; or (2) ignore the issue in the formalized analysis design and use an ad-hoc strategy in the analysis implementation.

Our proposed widening operator is a means to formalize control-flow sensitivity in a manner that guarantees soundness, but does not require that a sensitivity to be baked into the semantics. On a practical level, it also allows the analysis designer to experiment with many different sensitivities without modifying the analysis implementation.

2.2 Widening Operator

Our goal is to *limit* the number of states contained in the fixpoint, while still retaining soundness. We do so by defining a widening operator for the fixpoint computation, which acts on entire sets of states rather than on individual abstract domains inside the states. This widening operator: (1) partitions the current set of reachable states into disjoint sets; (2) merges all of the states in each partition into a single state that over-approximates that partition; and (3) unions the resulting states together into a new set that contains only a single state per partition. The widening operator controls the performance and precision of the analysis by setting a bound on the number of states allowed: there can be at most one state per partition. Decreasing the number of partitions can speed up the fixpoint computation, thus helping performance, but can also merge more states together in each partition, thus hindering precision.

Formally, the widening operator for control-flow sensitivity is parameterized by a (unspecified) equivalence relation $\sim$ on abstract states. Given a widening operator $\triangledown$ on individual abstract domains, our new widening operator $\triangledown^\sharp$ is defined as:

$$\triangledown^\sharp \in \mathcal{P}(\Sigma^\sharp) \times \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp)$$
$$A \triangledown^\sharp B = \left\{ \bigtriangledown_{\hat{\varsigma} \in X} \hat{\varsigma} \;\middle|\; X \in (A \cup B)/{\sim} \right\}$$

where for a set S the notation $S/{\sim}$ means the set of partitions of S according to equivalence relation $\sim$, and the widening operator $\triangledown$ on individual abstract domains is used to merge the states in each resulting partition into a single state. Note that if the number of partitions induced by $\sim$ is finite, then the number of states in each partition is also finite because we apply the widening operator at each step of the fixpoint computation.

Theorem 1 (WIDENING). *If the number of partitions induced by $\sim$ is finite, then $\triangledown^\sharp$ is a widening operator.*

Proof. Follows from the definition of a widening operator [7].

We now lift the transition relation $\mathcal{F}^\sharp$ in a similar fashion as before, except instead of using set union we use our widening operator: $\overset{\triangledown}{\mathcal{F}}{}^\sharp(S) \overset{\text{def}}{=} S \ \triangledown^\sharp \ \mathcal{F}^\sharp(S)$. Then the control-flow sensitive abstract semantics is defined as $[\![P]\!]^\sharp_\triangledown \overset{\text{def}}{=} \mathsf{lfp}_{\Sigma^\sharp_I} \overset{\triangledown}{\mathcal{F}}{}^\sharp$.

Even though we have not specified the equivalence relation that parameterizes the widening operator, we can still prove the soundness of the analysis. Informally, because the widening operator merges the states within each partition using $\triangledown$, the reachable states using $\overset{\triangledown}{\mathcal{F}}{}^\sharp$ over-approximate the reachable states using $\overset{\circ}{\mathcal{F}}{}^\sharp$. Thus, the control-flow sensitive abstract semantics is sound with respect to the original abstract semantics:

Theorem 2 (SOUNDNESS)

$$\gamma([\![P]\!]^\sharp) \subseteq \gamma([\![P]\!]^\sharp_\triangledown)$$

Proof. We must show that (1) the least fixpoint denoted by $[\![P]\!]^\sharp_\triangledown$ exists; and (2) it over-approximates $[\![P]\!]^\sharp$.

1. The existence of the fixpoint follows from part 2 of the definition of a widening operator as given by Cousot and Cousot [7, def. 9.1.3.3].
2. That the widened fixpoint over-approximates the original fixpoint follows from part 1 of the definition of a widening operator as given by Cousot and Cousot [7, defs. 9.1.3.1–9.1.3.2].

2.3 Control-Flow Sensitivity

It remains to show how our widening operator determines the control-flow sensitivity of the analysis. The determining factor is how the states are partitioned, which is controlled by the specific equivalence relation on states $\sim$ that parameterizes the widening operator. The question is, what constitutes a good choice for the equivalence relation? For Theorem 1 to hold, it must induce a finite number of partitions, but what other characteristics should it have? Our goal is tractability with a minimal loss of precision; this means we should try to partition the states so that there are a tractable number of partitions *and* the states within each partition are as similar to each other as possible (to minimize the information lost to merging).

A reasonable heuristic is to partition states based on how those states were computed, i.e., the execution history that led to each particular state. The hypothesis is that if two states were derived in a similar way then they are more likely to be similar. This heuristic of similarity is exactly the one used by existing control-flow sensitivities, such as flow-sensitive maximal fixpoint, *k*-CFA, object-sensitivity, property simulation, etc. These sensitivities each compute an abstraction of the execution history (e.g., current program point, last *k* call-sites, last *k* allocation sites, etc.) and use that abstraction to partition and merge the states during the analysis.

Therefore, the widening operator should partition the set of states according to their control-flow sensitivity approximation:

$$\hat{\varsigma}_1 \sim \hat{\varsigma}_2 \iff \pi_{\hat{\tau}}(\hat{\varsigma}_2) = \pi_{\hat{\tau}}(\hat{\varsigma}_2)$$

where each state contains an *abstract trace* $\hat{\tau}$ describing some abstraction of the execution history, and $\pi_{\hat{\tau}}(\hat{\varsigma})$ projects a state's abstract trace. This definition causes the widening operator to merge all states with the same trace, i.e., all states with the same approximate execution history. The widened analysis can be defined without specifying a particular abstract trace domain; different trace domains can be plugged in after the fact to yield different sensitivities.

Trace Abstractions. We have posited that control-flow sensitivity is based on an abstraction of the execution history of a program, called a trace. This implies that the trace abstraction is related to the *trace-based* concrete collecting semantics, which contains all reachable execution paths, i.e., sequences of states, rather than just all reachable states. An abstract trace is an abstraction of a set of paths in the concrete collecting semantics. For example, a flow-sensitive trace abstraction records the current program point, abstracting all paths that reach that program point. A context-sensitive trace abstraction additionally records the invocation context of the current function, abstracting all paths that end in that particular invocation context (e.g., as in Nielson and Nielson's mementoes [21]). Different forms of context-sensitivity define the abstract "context" differently: for example, traditional k-CFA defines it as the last k call-sites encountered in the concrete trace; stack-based k-CFA considers the top k currently active (i.e., not yet returned) calls on the stack; object sensitivity considers abstract allocation sites instead of call-sites; and so on.

We note that it is not necessary for the trace abstraction to soundly approximate the concrete semantics for the resulting analysis to be sound. The trace abstraction is a heuristic for partitioning the states; as long as the number of elements in the trace abstraction domain is finite (and hence the number of partitions enforced by the widening operator is finite), the analysis will terminate with a sound solution. In fact, it isn't strictly necessary for $\sim$ to be based on control-flow at all—exploring other heuristics for partitioning states would be in interesting avenue for future work.

2.4 Semantic Requirements

To benefit from widening-based control-flow sensitivity, an abstract semantics must satisfy certain requirements. To abstract control, the analysis must be able to introduce new program execution paths that over-approximate existing execution paths. To make this possible, we argue that there should be some explicit notion in the program semantics of the "rest of the computation"—i.e., a continuation. When the analysis abstracts control, it is abstracting these continuations. The explicit control-flow representation can take a number of possible forms. For example, it could be in the form of a syntactic continuation (e.g., if a program is in continuation-passing style then the "rest of the computation" is given as a closure in the store) or a semantic continuation (e.g., the continuation stack of an abstract machine). Since the abstract states form a lattice, any two distinct states must have a join, and (according to our requirement) this joined state must contain a continuation that over-approximates the input states' continuations. Thus, by joining states the analysis approximates control as well as data.

Some forms of semantics do not meet this requirement, including various forms proposed as being good foundations for abstract interpretation [16,24,25]. For example, big-step and small-step structural operational semantics implicitly embed the continuations in the semantic rules. Direct-style denotational semantics similarly embeds this information in the translation to the underlying meta-language. This means that there is no way to abstract and over-approximate control-flow; the analysis must use whatever control-flow the original semantics specifies (or, alternatively, use ad-hoc strategies baked into the analysis implementation to silently handle control-flow sensitivity). Some limited forms of control-flow sensitivity may still be expressed when the analysis takes care to join only those states that already have the same continuation (e.g., flow-sensitive maximal fixpoint), but many other forms (e.g., *k*-CFA or other forms of context-[in]sensitivity) remain difficult to express.

3 Relating and Combining Sensitivities

One of the goals of this work is to make it easier for analysis designers to experiment with new trace abstractions. To this end, it would be useful to systematically create new trace abstractions from existing ones and to understand how trace abstractions relate to one another.

An obvious way to combine trace abstractions into a new form of control-flow sensitivity is to take their *product*.[3] Given two trace abstractions, one constructs their product by taking the cartesian product of the corresponding sets and defining the update function to act pairwise on the resulting tuple. A less obvious method of combining trace abstractions is to take their *sum*. This is a novel way to create new control-flow sensitivities that has not been presented before. Think of a trace abstraction as allowing the analysis to decide whether two abstract states computed during the analysis should be joined. Informally, the product of two trace abstractions joins two states only if *both* traces agree that the states should be joined, whereas the sum of two trace abstractions joins two states if *either* trace determines that the states should be joined.

In the next section, we describe how to construct the sum of two traces. We then show that sum and product are the join and meet operations of a lattice of control-flow sensitivities. This construction suggests new control-flow sensitivities that could be used in practice and also enables a fully automated exploration of control-flow sensitivities that complements manual exploration. The supplementary material contains an implementation of the product and sum operators described here, as part of the implementation of the example abstract semantics described in Section 4.

3.1 Sums of Trace Abstractions

While the product of two traces is obvious, constructing the sum is unintuitive. We formally define a trace abstraction as a (unspecified) finite set $\Theta^\sharp$, a distinct element $\mathbf{1} \in \Theta^\sharp$ that acts as an initial trace for the analysis, and a trace update function $\tau_{update} : (\Sigma^\sharp \times \Theta^\sharp) \uplus \mathbf{1} \to \Theta^\sharp$ that specifies how the trace changes at each statement transition in the abstract semantics. The pair $(\Theta^\sharp, \tau_{update})$ is the object we call a trace abstraction.

[3] Another interesting combination to explore would be the *reduced product*, however it is not in general possible to automatically derive the reduced product of two domains [7, §10].

When discussing multiple trace abstractions we use Θ_X, τ_X, and $\mathbf{1}_X$ to denote the $\Theta^\sharp$, τ_{update}, and initial trace for each trace abstraction X.

A naïve attempt to construct the sum would use the disjoint union of the trace abstractions' underlying sets. However, this attempt fails because each trace abstraction has a unique initial trace, and thus the disjoint union does not constitute a valid trace abstraction. To create a valid sum $X + Y$ from trace abstractions X and Y, we must create a new set Θ_{X+Y} whose initial trace $\mathbf{1}_{X+Y} \in \Theta_{X+Y}$ "agrees" with both $\mathbf{1}_X$ and $\mathbf{1}_Y$, in a sense that we formalize below. The sum transition function τ_{X+Y} must also "agree" with both τ_X and τ_Y, in the same sense. The central insight behind the sum construction is to construct an equivalence relation between elements of Θ_X and Θ_Y, and let the elements of Θ_{X+Y} be the corresponding equivalence classes. Then $\mathbf{1}_{X+Y}$ and τ_{X+Y} agree with the individual trace abstractions X and Y if they produce equivalence classes that contain the same elements that would have been produced by X and Y individually. It remains to describe how the analysis creates these equivalence classes: they cannot be constructed before the analysis begins, rather the analysis constructs them dynamically (i.e., as it executes) in the following way.

Definition 1. *Let X and Y be trace abstractions and $\hat{\varsigma}$ be an abstract state. Inductively define an equivalence relation $\sim$ on the disjoint union $\Theta_X \uplus \Theta_Y$ by taking the symmetric, reflexive and transitive closure while applying the following rules:*

$$\mathbf{1}_X \sim \mathbf{1}_Y$$

$$\frac{i, j \in \{X, Y\} \qquad a \in \Theta_i \qquad b \in \Theta_j \qquad a \sim b}{\tau_i(\hat{\varsigma}, a) \sim \tau_j(\hat{\varsigma}, b)}$$

The sum $X + Y$ has underlying set $\Theta_{X+Y} = (\Theta_X \uplus \Theta_Y)/\!\sim$, i.e., the set of equivalence classes of $\Theta_X \uplus \Theta_Y$ according to $\sim$. The equivalence relation is defined by construction so that τ_X and τ_Y will always agree on which equivalence class of $\hat{\tau}_X \uplus \hat{\tau}_Y$ to transition to. We now define the disjoint union transition function τ_{X+Y}. For $\hat{\tau} \in \Theta_X$, let $[\hat{\tau}]$ denote the equivalence class of $\hat{\tau}$ in Θ_{X+Y}. The function τ_{X+Y} is defined as follows: for any $\hat{\varsigma} \in \Sigma^\sharp$ and $[\hat{\tau}] \in \Theta_{X+Y}$, pick some $\hat{\tau}' \in \Theta_X$ so that $\hat{\tau}' \in [\hat{\tau}]$. Then $\tau_{X+Y}(\hat{\varsigma}, [\hat{\tau}]) = [\tau_X(\hat{\varsigma}, \hat{\tau}')]$. The equivalence relation ensures this is well defined for any valid choice of $\hat{\tau}'$. By symmetry, the same applies to $\hat{\tau} \in \Theta_Y$.

The essence of this definition is that it causes the following diagram to commute, making τ_X and τ_Y agree with τ_{X+Y}:

$$\begin{tikzcd}
\Sigma^\sharp \times \Theta_X \uplus \mathbf{1} \arrow[r, "{\substack{(\hat{\varsigma},x) \mapsto (\hat{\varsigma},[x]) \\ \mathbf{1} \mapsto \mathbf{1}}}"] \arrow[d, "\tau_X"] & \Sigma^\sharp \times \Theta_{X+Y} \uplus \mathbf{1} \arrow[d, "\tau_{X+Y}"] & \Sigma^\sharp \times \Theta_Y \uplus \mathbf{1} \arrow[l, "{\substack{(\hat{\varsigma},[y]) \mapsfrom (\hat{\varsigma},y) \\ \mathbf{1} \mapsfrom \mathbf{1}}}"'] \arrow[d, "\tau_Y"] \\
\Theta_X \arrow[r, "{[\cdot]}"'] & \Theta_{X+Y} & \Theta_Y \arrow[l, "{[\cdot]}"]
\end{tikzcd}$$

$X + Y$ is a trace abstraction that is less precise than both X and Y individually, but still contains some information from both. When implemented, the definition of the equivalence relation is unknown until runtime, where it is incrementally discovered by the analysis. Initially, the relation is the one forcing $\mathbf{1}_X \sim \mathbf{1}_Y$. At each iteration of the fixpoint calculation the functions τ_X and τ_Y are computed, and the results are used to discover more equivalences.

From another perspective, one can also view a trace abstraction X as a finite automaton with a set of automaton states Θ_X, a transition function τ_X, and an initial point $\mathbf{1}_X$. The input alphabet is the set $\Sigma^\sharp$. We were surprised to discover that, from this perspective, our definition of summing trace abstractions corresponds exactly to a widening operator on finite automata described by Bartzis and Bultan [4]. Their operator was designed to provide a "less precise" finite automaton that accepts a larger language than both of its inputs.

3.2 Trace Abstractions Form a Lattice

The sum and product operations described above are dual to each other in a special way: they are the join and meet operations of a lattice. The lattice partial order is based on the precision of the trace abstractions. Using the notation from the previous section, we say that a trace abstraction X is *more precise* than Y (written as $X \leq Y$) if there exists a relation on the corresponding automata satisfying certain properties.

Definition 2. *$X \leq Y$ if there is a relation $R \subset Y \times X$ such that*

1. $(\mathbf{1}_Y, \mathbf{1}_X) \in R$
2. $(y, x) \in R$ *implies for all* $\hat{\varsigma} \in \Sigma^\sharp$, $(\tau_Y(\hat{\varsigma}, y), \tau_X(\hat{\varsigma}, x)) \in R$
3. *R is injective, meaning* $(y, x) \in R$ *and* $(y', x) \in R$ *implies* $y = y'$.

The relation R forces Y and X to behave in the same way, but also requires that X has "more" members than Y. It can be likened to an injective function. The relation $\leq$ is a preorder; by implicitly taking equivalence classes it becomes a partial order. Intuitively, $X < Y$ corresponds to the intuition "X is strictly more precise than Y", in every way of measuring it. Members of the same equivalence class correspond to families of trace abstractions that provide exactly the same precision.

Theorem 3. *The space of trace abstractions form a lattice, where sum corresponds to join and product corresponds to meet.*

Proof. The proof follows from elementary results in category theory, order theory and our definitions. The details are given in the supplementary materials.

Use of Category Theory. Category theory provides useful constructions that apply in general settings. We arrived at our construction of the sum operator and the lattice of sensitivities via category theory, because they were non-obvious without this perspective. We used category theory to derive the definition for sums of trace abstractions and prove the theorems elegantly, and we suspect it can be used to achieve further insights into combining sensitivities. In our supplementary material we detail how we used it to arrive at our results.

4 Analysis Design Example

In this section, we give a detailed example of how to build an abstract interpreter with separate and tunable control-flow sensitivity. Our example picks up in the middle of the design process: an analysis designer has formally defined an abstract semantics that is amenable to defining control-flow sensitivity (cf. Section 2) and the semantics has been proven sound with respect to a concrete semantics.[4] We show how to extend this analysis to support tunable control flow and how to easily and modularly tune the control-flow sensitivity of the resulting analysis. The supplementary material contains an implementation of this example analysis written in Scala.

4.1 Syntax

Figure 1 gives the syntax of a small but featureful language. It contains integers, higher-order functions, conditionals, and mutable state. As discussed in Section 2.4, tunable control-flow sensitivity requires an explicit representation of a program's control. For this example we chose to make control-flow explicit in the program syntax, hence we use continuation-passing style (CPS). We assume that the CPS syntax is the result of a CPS-translation from a programmer-facing, direct-style syntax.

$$n \in \mathbb{Z} \qquad x \in \mathit{Variable} \qquad \oplus \in \mathit{BinaryOp} \qquad \ell \in \mathit{Label}$$

$$\begin{aligned}
L \in \mathit{Lam} &::= \lambda \overrightarrow{x}\,.\,S \\
T \in \mathit{Trivial} &::= [n_1..n_2] \;\mid\; x \;\mid\; L \;\mid\; T_l \oplus T_r \\
S \in \mathit{Serious} &::= \textsf{let}\ x = T\ \textsf{in}\ S \;\mid\; \textsf{set}\ x = T\ \textsf{in}\ S \;\mid\; \textsf{if}\ T\ S_t\ S_f \;\mid\; x(\overrightarrow{T})
\end{aligned}$$

Fig. 1. Continuation-passing style (CPS) syntax for the example language. Vector notation denotes an ordered sequence. The notation $[n_1..n_2]$ denotes nondeterministic choice from a range of integers, e.g., to simulate user input.

As usual for CPS [22], expressions are separated into two categories: *Trivial* and *Serious*. *Trivial* expressions are guaranteed to terminate and to have no side-effects; *Serious* expressions make no such guarantees. Functions take an arbitrary number of arguments and can represent either user-defined functions from the direct-style program (modified to take an additional continuation parameter) or the continuations created by the CPS transform (including a **halt** continuation that terminates evaluation). We assume that it is possible to syntactically disambiguate among calls to user-defined functions, calls to continuations that correspond to a function return, and all other calls. All syntactic entities have an associated unique label $\ell \in \mathit{Label}$; the expression $\cdot^{\ell}$ retrieves this label (for example, the label of *Serious* expression S is S^{ℓ}).

4.2 Original Abstract Semantics

The original abstract semantics defines a computable approximation of a program's behavior using a small-step abstract machine. Figure 2a describes the semantic domains

[4] For reasons of space we omit the concrete semantics and soundness proof; neither are novel.

(for now, ignore the boxed elements[5]). An abstract state consists of a set of *Serious* expressions $\overline{S}$ (which represents the set of expressions that might execute at the current step), an abstract environment $\hat{\rho}$, and an abstract store $\hat{\sigma}$. Because the language is dynamically typed, any variable may be an integer or a closure at any time. Thus, abstract values are a product of two abstractions: one for integers and one for closures.[6] Integers are abstracted with the constant propagation lattice $\mathbb{Z}^\sharp = \mathbb{Z} \cup \{\top_{\mathbb{Z}^\sharp}, \bot_{\mathbb{Z}^\sharp}\}$, and closures are abstracted with the powerset lattice of abstract closures. The analysis employs a finite address domain $Address^\sharp$ and a function *alloc* to generate abstract addresses; we leave these elements unspecified for brevity.

Figure 2b describes the semantic function $\hat{\eta}$, which abstractly interprets *Trivial* expressions. Literals evaluate to their abstract counterparts, injected into a tuple. Variable lookup joins all the abstract values associated with that variable. Figure 2c describes the abstract semantics of *Serious* expressions. In the rule for function calls, we use the notation $\overrightarrow{T}$ to mean the sequence of argument expressions, T_i to mean a particular argument expression, and $[\overrightarrow{p_i \rightarrow q_i}]$ to mean each p_i maps to its corresponding q_i. Note the sources of non-determinism: An **if** statement may lead to multiple states (i.e., when the condition's abstract value is not precise enough to send the abstract interpretation down only one branch). A function call also may lead to multiple states (i.e., when evaluating the function's name leads to a set of closures, each of which is traced by the abstract interpretation). The semantics employs *weak updates*: when a value is updated, the analysis joins the new value with the old value. It is possible under certain circumstances to strongly update the store (by replacing the old value instead of joining with it), but for simplicity our example always uses weak updates.

The full analysis is defined as the reachable states abstract collecting semantics of *Serious* evaluation: $[\![S]\!]^\sharp = \mathbf{lfp}_{\Sigma^\sharp_I} \mathring{\mathcal{F}}^\sharp$. This analysis is sound and computable, however it is still intractable because the set of reachable states grows exponentially with the number of nondeterministic branch points. To make this analysis tractable requires some form of control-flow sensitivity.

4.3 Tunable Control-Flow Sensitivity

We now extend the abstract semantics for our language to express tunable control-flow sensitivity. As described in Section 2, we extend the definition of abstract states to include a trace abstraction domain $\hat{\tau} \in \Theta^\sharp$. We leave the trace abstraction domain unspecified; the specific instantiation of the trace abstraction domain will determine the analysis control-flow sensitivity.

We make three changes to the abstract semantics of the previous section to integrate trace abstractions into the semantics; these are represented by the boxed elements in Figure 2. First, we add the trace abstraction domain to the abstract state definition. Next, we modify $\mathcal{F}^\sharp$ to operate on this new domain. This change gives the trace update mechanism access to all the data needed to compute a new abstract trace. Finally, we

[5] All unboxed elements define the original abstract semantics. Boxed elements describe the extensions that support parameterized control-flow sensitivity; they are described in Section 4.3.

[6] For brevity, we omit error-handling semantics and sometimes omit one part of the tuple, when the meaning is clear from the context (e.g., when interpreting *Serious* values in Figure 2c).

$$\hat{n} \in \mathbb{Z}^\sharp \qquad \hat{\oplus} \in \mathit{BinaryOp}^\sharp \qquad \boxed{\hat{\tau} \in \Theta^\sharp}$$

$$\begin{array}{rcll}
\hat{\varsigma} \in \Sigma^\sharp & = & \mathcal{P}(\mathit{Serious}) \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \times \boxed{\Theta^\sharp} & \text{(abstract states)} \\
\hat{\rho} \in \mathit{Env}^\sharp & = & \mathit{Variable} \to \mathcal{P}(\mathit{Address}^\sharp) & \text{(environments)} \\
\hat{\sigma} \in \mathit{Store}^\sharp & = & \mathit{Address}^\sharp \to \mathit{Value}^\sharp & \text{(stores)} \\
\widehat{clo} \in \mathit{Closure}^\sharp & = & \boxed{\Theta^\sharp} \times \mathit{Env}^\sharp \times \mathit{Lam} & \text{(closure values)} \\
\hat{v} \in \mathit{Value}^\sharp & = & \mathbb{Z}^\sharp \times \mathcal{P}(\mathit{Closure}^\sharp) & \text{(abstract values)} \\
\mathcal{F}^\sharp \in \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp) & = & \text{Figure 2c} & \text{(transition function)}
\end{array}$$

(a) Abstract semantic domains.

$$\hat{\eta} \in \mathit{Trivial} \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \times \boxed{\Theta^\sharp} \to \mathit{Value}^\sharp$$

$$\begin{aligned}
\hat{\eta}([n_1..n_2], \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \langle \alpha([n_1..n_2]), \emptyset \rangle \\
\hat{\eta}(x, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \bigsqcup_{\hat{a} \in \hat{\rho}(x)} \hat{\sigma}(\hat{a}) \\
\hat{\eta}(\lambda \vec{x} \,.\, S, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \langle \bot_{\mathbb{Z}^\sharp}, \{\langle \boxed{\hat{\tau}}, \hat{\rho}, \lambda \vec{x} \,.\, S \rangle\} \rangle \\
\hat{\eta}(T_l \oplus T_r, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \hat{\eta}(T_l, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \mathbin{\hat{\oplus}} \hat{\eta}(T_r, \hat{\rho}, \hat{\sigma}, \hat{\tau})
\end{aligned}$$

(b) Abstract *Trivial* evaluation.

$S_i \in \overline{S}$	*where*	S'	$\hat{\rho}'$	$\hat{\sigma}'$	$\boxed{\hat{\tau}'}$
let $x = T$ **in** S_b	$[\![T]\!] = \hat{v}$	S_b	$\hat{\rho}[x \mapsto \hat{a}']$	$\hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{v}]$	$\boxed{\tau_{stmt}(\hat{\varsigma}, S_b)}$
set $x = T$ **in** S_b	$[\![T]\!] = \hat{v} \wedge \hat{\rho}(x) = \vec{\hat{a}}$	S_b	$\hat{\rho}$	$\hat{\sigma} \sqcup [\overrightarrow{\hat{a}_i \mapsto \hat{v}}]$	$\boxed{\tau_{stmt}(\hat{\varsigma}, S_b)}$
if $T\ S_t\ S_f$	$[\![T]\!] \notin \{\hat{0}, \bot_{\mathbb{Z}^\sharp}\}$	S_t	$\hat{\rho}$	$\hat{\sigma}$	$\boxed{\tau_{stmt}(\hat{\varsigma}, S_t)}$
	$[\![T]\!] \sqsupseteq \hat{0}$	S_f	$\hat{\rho}$	$\hat{\sigma}$	$\boxed{\tau_{stmt}(\hat{\varsigma}, S_f)}$
$\mathrm{x}(\vec{T})$	$[\![T_i]\!] = \hat{v}_i \quad \wedge$ $\underbrace{\langle \boxed{\hat{\tau}_c}, \hat{\rho}_c, \lambda \mathbf{y} \,.\, S_c \rangle}_{\widehat{clo}} \in [\![x]\!]$	S_c	$\hat{\rho}_c[\overrightarrow{y_i \mapsto \hat{a}'_i}]$	$\hat{\sigma} \sqcup [\overrightarrow{\hat{a}'_i \mapsto \hat{v}_i}]$	$\boxed{\tau_{call}(\hat{\varsigma}, \widehat{clo})}$

(c) Abstract transition function $\mathcal{F}^\sharp$, where $[\![\cdot]\!] = \hat{\eta}(\cdot, \hat{\rho}, \hat{\sigma}, \hat{\tau})$ and a fresh address $\hat{a}'$ is given by *alloc*. Given a current state $\hat{\varsigma} = \langle \overline{S}, \hat{\rho}, \hat{\sigma}, \hat{\tau} \rangle$, the transition function yields a set of new states $\mathcal{F}^\sharp(\hat{\varsigma}) = \overline{\langle \{S'\}, \hat{\rho}', \hat{\sigma}', \hat{\tau}' \rangle}$.

Fig. 2. A standard abstract semantics over a simple abstract value domain (constant- and closure-propagation). The $\boxed{\text{boxed}}$ elements indicate what extensions are necessary for this semantics to support parameterized control-flow sensitivity.

extend abstract closures to contain an abstract trace. Intuitively, a closure's abstract trace corresponds to the trace that existed before a function was called. Any analysis that tracks calls and returns (e.g., stack-based k-CFA) can use this extra information to simulate stack behavior upon exiting a function call by restoring the trace to the point before a function was called.

The analysis designer tunes control-flow sensitivity by specifying an abstract trace domain and a pair of transition functions that generate new abstract traces:

$$\begin{aligned} \hat{\tau} &\in \Theta^\sharp \\ \tau_{stmt} &\in \Sigma^\sharp \times Serious \to \Theta^\sharp \\ \tau_{call} &\in \Sigma^\sharp \times Closure^\sharp \to \Theta^\sharp \end{aligned}$$

The abstract trace domain summarizes the history of program execution. The abstract trace transition function τ_{stmt} specifies how to generate a trace when execution transitions between two program points in the same function. The abstract trace transition function τ_{call} specifies how to generate a trace when execution transitions across a function call. The program analysis is defined as the widened reachable states abstract collecting semantics, $[\![S]\!]^\sharp_\nabla = \mathbf{lfp}^{\nabla}_{\Sigma^\sharp_I} \mathcal{F}^\sharp$, where the equivalence relation $\sim$ is the one given in Section 2.3.

The precision and performance of the analysis depend on the particular choice of trace abstraction for control-flow sensitivity. In the remainder of this section, we present six illustrative examples.

Flow-Insensitive, Context-Insensitive Analysis. In a flow-insensitive analysis, any *Ser-ious* expression can execute after any other *Serious* expression, regardless of where those expressions appear in the program. Rather than compute separate solutions for each program point, the analysis computes a single solution for the entire program. Using our method, the analysis designer can specify flow-insensitive analysis by making the $\Theta^\sharp$ domain a single value, so that all states will necessarily have the same abstract trace and hence belong to the same partition.

Algorithm 1. Flow-insensitive, context-insensitive

$$\begin{aligned} \hat{\tau} \in \Theta^\sharp &= \mathbf{1} \\ \tau_{stmt}(\_, \_) &= \mathbf{1} \\ \tau_{call}(\_, \_) &= \mathbf{1} \end{aligned}$$

Flow-Sensitive (FS), Context-Insensitive Analysis. A flow-sensitive analysis executes statements in program-order, computing a single solution for each program point. The analysis designer can specify flow-sensitive analysis by making the $\Theta^\sharp$ domain the set of program labels and updating the trace at each step to be the current program point. The abstract semantics at each step collects all states at the same program point and joins them together, constraining the maximum number of abstract states to be the number of program points. In the dataflow analysis community this is called the flow-sensitive *maximal fixpoint analysis* (MFP).

Algorithm 2. Flow-sensitive, context-insensitive

$$\begin{aligned} \hat{\tau} \in \Theta^\sharp &= \mathit{Label} \\ \tau_{stmt}(\_, S) &= S^\ell \\ \tau_{call}(\_, \langle \_, \_, \lambda \mathbf{y} \,.\, S_c\rangle) &= S_c^\ell \end{aligned}$$

FS + Traditional k-CFA Analysis. Traditional k-CFA [26] is a context-sensitive analysis that keeps track of the last k call-sites encountered along an execution path and uses this *callstring* to distinguish information at a given program point that arrives via different routes. At each function call, the analysis appends the call-site to the callstring and truncates the result so that the new callstring has at most k elements. The analysis designer can specify flow-sensitive k-CFA by making the $\Theta^\sharp$ domain contain a tuple of the current program point and the callstring (as a sequence of labels). The first element of the tuple tracks flow-sensitivity; the second element of the tuple tracks context-sensitivity. Note that the current callstring is left unmodified when returning from a call (i.e., calling the continuation that was passed into the current function); thus the callstring does *not* act like a stack.

Algorithm 3. Flow-sensitive, k-CFA

$$\begin{aligned} \hat{\tau} \in \Theta^\sharp &= \mathit{Label} \times \mathit{Label}^\star \\ \tau_{stmt}(\langle \_, \_, \_, \hat{\tau}\rangle, S) &= \langle S^\ell, \pi_2(\hat{\tau})\rangle \\ \tau_{call}(\langle S, \_, \_, \hat{\tau}\rangle, \langle \_, \_, \lambda \mathbf{y} \,.\, S_c\rangle) &= \langle S_c^\ell, \hat{\tau}'\rangle \end{aligned}$$

$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (S^\ell :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

Flow-Sensitive, Stack-Based k-CFA Analysis. In dataflow analysis, k-CFA is usually defined as having a stack-like behavior: upon returning from a function call, the current callstring is discarded and replaced by the callstring that held immediately before making that function call (in effect, the callstring is "pushed" when entering a function and "popped" when exiting the function). The analysis designer can achieve this behavior by modifying τ_{call} to detect continuation calls that correspond to function returns and to replace the current callstring with the callstring held in the return continuation's closure. The CPS transformation guarantees this callstring to be the one that held immediately before the current function was called.

Analysis Example 4. Flow-sensitive, stack-based k-CFA

$$\hat{\tau} \in \Theta^\sharp = \mathit{Label} \times \mathit{Label}^\star$$
$$\tau_{stmt}(\langle \_, \_, \_, \hat{\tau}\rangle, S) = \langle S^\ell, \pi_2(\hat{\tau})\rangle$$
$$\tau_{call}(\langle S, \_, \_, \hat{\tau}\rangle, \langle \hat{\tau}_c, \_, \lambda \boldsymbol{y} \,.\, S_c\rangle) = \langle S_c^\ell, \hat{\tau}'\rangle$$

$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (S^\ell :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}_c) & \text{if } S \in \mathit{ReturnKont} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

FS + k-allocation-site Sensitive Analysis. Object-sensitivity [20] is a popular form of context-sensitive control-flow sensitivity for object-oriented languages. We do not have objects in our example language, but as noted elsewhere [27] object-sensitivity should more properly be termed *allocation-site* sensitivity—it defines a function's context in terms of the last k allocation-sites (i.e., abstract addresses) rather than callstrings. Under the assumption that every function call uses a variable as the first argument, the analysis designer can employ a form of allocation-site sensitivity by using that variable's address to form the abstract trace. The value **self** refers to the address of the call's first argument. In an object-oriented language, this argument always corresponds to the receiver of a method (i.e., the **self** or **this** pointer).

Analysis Example 5. Flow-sensitive, k-allocation-site sensitive

$$\hat{\tau} \in \Theta^\sharp = \mathit{Label} \times \mathit{Address}^{\sharp\star}$$
$$\tau_{stmt}(\langle \_, \_, \_, \hat{\tau}\rangle, S) = \langle S^\ell, \pi_2(\hat{\tau})\rangle$$
$$\tau_{call}(\langle S, \_, \_, \hat{\tau}\rangle, \langle \hat{\tau}_c, \_, \lambda \boldsymbol{y} \,.\, S_c\rangle) = \langle S_c^\ell, \hat{\tau}'\rangle$$

$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (\textbf{self} :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}_c) & \text{if } S \in \mathit{ReturnKont} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

Property Simulation Analysis. A more unusual form of control-flow sensitivity is Das et al.'s property simulation [9]. Property simulation relies on a finite-state machine

(FSM) that describes a higher-level notion of execution trace—for example, an FSM whose states track whether a file is open or closed, or whether a lock is locked or unlocked. The analysis transitions this FSM according to the instructions it encounters. At a join point in the program (e.g., immediately after the two branches of a conditional) the analysis either merges the execution state or not depending on whether the FSMs along the two paths are in the same FSM state. The analysis designer can specify property simulation by making $\Theta^\sharp$ be a tuple that contains the current program point and the current state of the FSM. The FSM is updated based on an API (e.g., for file or lock operations) so that τ_{call} will transition the FSM accordingly.

Analysis Example 6. Flow-sensitive, property-sensitive

$$\begin{aligned} \hat{\tau} \in \Theta^\sharp &= Label \times FSM \\ \tau_{stmt}(\langle \_, \_, \_, \hat{\tau}\rangle, S) &= \langle S^\ell,\ \pi_2(\hat{\tau})\rangle \\ \tau_{call}(\langle S, \_, \_, \hat{\tau}\rangle, \langle \hat{\tau}_c, \_, \lambda \mathbf{y} \,.\, S_c\rangle) &= \langle S_c^\ell,\ \delta_{FSM}(S, \pi_2(\hat{\tau}))\rangle \end{aligned}$$

5 Related Work

In abstract interpretation, there is a relatively small but dedicated body of research on trace abstraction and on formalizing control-flow sensitivity as partitioning. What distinguishes our work from most prior efforts is a different focus: prior work focuses on the *integration* of control-flow abstractions into an existing analysis; our work focuses on the *separation* of control-flow abstractions from an existing analysis, so that it is easier for analysis designers to experiment with different control-flow sensitivities. In this section, we discuss the implications of these differences. Broadly, no prior work has couched control-flow sensitivity in terms of a widening operator based on abstractions of the program history, which permits a simpler, more general, and more tunable formulation of control-flow sensitivity.

A Posteriori Soundness. Our work is most similar to Might and Manolios' *a posteriori* soundness for non-deterministic abstract interpretation [19], which also seeks to separate the aspects of an analysis that affect its precision from those that affect its soundness. Both techniques achieve this separation by introducing a level of indirection, although the mechanisms are different. Our technique uses an equivalence relation that partitions abstract states. Might and Manolios' uses an *abstract allocation policy* that can dynamically allocate the resources that determine how to partition abstract states. We accomplish soundness by leveraging the soundness of widening. Might and Manolios accomplish soundness via their technique of an *a posteriori* proof: their abstract allocation policies induce a non-deterministic abstract semantics that can be shown to produce sound analysis results, even though the abstract semantics do not conform to the traditional simulation of the concrete semantics. Our work also re-formulates one of Might and Manolios' insights: that most control-flow (or heap-allocation) approximations are already sound because they add only extra information to the analysis.

A particular strength of Might and Manolios' work is that it makes it easy to express sound, adaptable analyses. A particular strength of our work is that it makes it easy to declaratively describe many forms of analyses and to systematically combine them. It is not clear whether the two techniques are equally expressive, nor whether they are equally useful in practice. An interesting line of research would be to explore how well each technique is suited to the practical discovery, design, and implementation of precise analyses and how the two techniques might compete with or complement each other.

Trace Partitioning. Our work is similar in some respects to the trace partitioning work by Mauborgne and Rival [15,23], which itself builds on the abstraction-by-partitioning of control-flow by Handjieva and Tzolovski [11]. Trace partitioning was developed in the context of the ASTRÉE static analyzer [5] for a restricted subset of the C language, primarily intended for embedded systems. Mauborgne and Rival observe that usually abstract interpreters are (1) based on reachable states collecting semantics, making it difficult to express control-flow sensitivity; and (2) designed to silently merge information at control-flow join points[7]—what in dataflow analysis is called "flow-sensitive maximal fixpoint" [12]. They propose a method to postpone these silent merges when doing so can increase precision; effectively they add a controlled form of path-sensitivity. They formalize their technique as a series of Galois connections.

Mauborgne and Rival describe a denotational semantics-based analysis that can use three criteria to determine whether to merge information at a particular point: the last k branch decisions taken (i.e., whether an execution path took the *true* or *false* branch); the last k while-loop iterations (effectively unrolling the loop k times); and the value of some distinguished variable. These criteria are guided by syntactic hints inserted into a program prior to analysis; the analysis itself can choose to ignore these hints and merge information regardless, as a form of widening. This feature is a form of *dynamic partitioning*, where the choice of partition is made as the analysis executes. Our sum abstraction (Section 3.1) is another form of dynamic partitioning.

The analysis described by Mauborgne and Rival requires that the program is non-recursive; it fully inlines all procedure calls to attain complete context-sensitivity. Because the semantics they formulate does not contain an explicit representation of continuations, there is no way in their described system to achieve other forms of context-sensitivity (e.g., k-CFA, including 0-CFA, i.e., context-insensitive analysis) without heavily modifying their design, implementation, and formalisms (cf. our discussion in Section 2.4). Because our method seeks more generality, it can express all of the sensitivities described by Mauborgne and Rival.

Predicate Abstraction. Fischer et al. [10] propose a method to join dataflow analysis with predicate abstraction using *predicate lattices* to gain a form of tunable intra-procedural path-sensitivity. At a high level these predicate lattices perform a similar "partition and merge" strategy as our own method. However, our method is more general: we can specify many more forms of control-flow sensitivity due to our insights

[7] By which they mean that the abstract semantics say nothing about merging information, but the implementation does so anyway.

regarding explicit control state. One can consider their work as a specific instantiation of our method using predicates as the trace abstraction. On the other hand, Fisher et al. use predicate refinement to *automatically* determine the set of predicates to use, which is outside the current scope of our method. In order to do the same, our method would need to add a predicate refinement strategy.

Context Sensitivity. There are several papers that describe various abstract interpretation-based approaches to specific forms of context sensitivity, including Nielson and Nielson [21], Ashley and Dybvig [1], Van Horn and Might [28], and Midtgaard and Jensen [17,18]. Nielson and Nielson describe a form of context-sensitivity based on abstractions of the history of a program's calls and returns [21]. Although this formulation is separable, it is not as general as the one described in this paper. For example, it cannot capture calls and returns in obfuscated binaries (which may contain no explicit calls and returns); to capture such behavior, a different formulation similar to property simulation is required [14]. Our parameterized, widening-based approach we describe is general enough to capture *either* of these formulations (and many more).

Ashley and Dybvig [1] give a reachable states collecting semantics formulation of k-CFA for a core Scheme-like language; they instrument both the concrete and abstract semantics with a *cache* that collects CFA information. The analysis as described in the paper is intractable (i.e., although it yields the same precision as k-CFA, the number of states remains exponential in the size of the program). Ashley and Dybvig implement a tractable, flow-insensitive version of the analysis independently from the formally-derived version, rather than deriving the tractable version directly from the formal semantics.

Van Horn and Might [28] also give a method for constructing analyses, using an abstract machine-based, reachable states collecting semantics of the lambda calculus. Their analysis includes a specification of k-CFA. An important contribution of their paper is a technique to abstract the infinite domains used for environments and semantic continuations using store allocation (this is an alternative we could have used for our example analysis in Section 4 instead of CPS form). As with Ashley and Dybvig, the analysis as described in their paper does not directly yield a tractable analysis. Van Horn and Might describe a tractable version of their analysis (not formally derived from the language semantics) that uses a single, global store to improve efficiency, but disallows flow-sensitive analysis because it computes a single solution for the entire program.

Midtgaard and Jensen [17] derive a tractable, demand-driven 0-CFA analysis for a core Scheme-like language using abstract interpretation. Their technique specifically targets 0-CFA, rather than general k-CFA. They employ a series of abstractions via Galois connections, the composition of which leads to the final 0-CFA analysis. In a later paper, Midtgaard and Jensen derive another 0-CFA analysis to compute both call *and* return information [18]. Our example semantics of Section 4 bears a resemblance to Midtgaard and Jensen's (and to van Horn and Might's machine construction), but our goals differ. Our example illustrates how to achieve a sound analysis with *arbitrary* control-flow sensitivity, without having to derive the soundness for each sensitivity.

6 Conclusions and Future Work

We have presented a method for program analysis design and implementation that allows the analysis designer to parameterize over control-flow abstractions. This separation of concerns springs from a novel theoretical insight that control-flow sensitivity is induced by a widening operator parameterized by trace abstractions. Our method makes it easier for an analysis designer to specify, implement, and experiment with many forms of control-flow sensitivity, which is critical for developing new, practical analyses. Our perspective on the space of trace abstractions as a category also enabled new insights into automatically constructing and combining trace abstractions in novel ways to achieve new forms of control-flow sensitivity. Our future work involves exploring these ideas further, for example, using combinatorial optimization to explore the vast space of possible trace abstractions. Additionally, our method applies not only to control-flow but to *any* property of a program that can be abstracted and that might be useful to partition the analysis state-space.

Acknowledgments. We are grateful to Matt Might, Jan Midtgaard, David van Horn, and Yannis Smaragdakis for their helpful advice and discussions. We are also grateful to the anonymous VMCAI reviewers for their feedback. This work was supported by NSF CCF-1117165.

References

1. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Transactions on Programming Languages and Systems (TOPLAS) 20(4) (July 1998)
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2001)
3. Banning, J.P.: An efficient way to find the side effects of procedure calls and the aliases of variables. In: ACM SIGPLAN Symposium on Principles of Programming Languages, POPL (1979)
4. Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 321–333. Springer, Heidelberg (2004)
5. 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)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM SIGPLAN Symposium on Principles of Programming Languages, POPL (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM SIGPLAN Symposium on Principles of Programming Languages, POPL (1979)
8. Cousot, P., Cousot, R.: Invited Talk: Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis of Functional Languages), invited paper
9. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2002)

10. Fischer, J., Jhala, R., Majumdar, R.: Joining dataflow with predicates. In: European Software Engineering Conference (2005)
11. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 200–214. Springer, Heidelberg (1998)
12. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7 (1977)
13. Kildall, G.A.: A unified approach to global program optimization. In: ACM SIGPLAN Symposium on Principles of Programming Languages, POPL (1973)
14. Lakhotia, A., Boccardo, D.R., Singh, A., Manacero, A.: Context-sensitive analysis of obfuscated x86 executables. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM (2010)
15. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
16. Metayer, D.L., Schmidt, D.: Structural operational semantics as a basis for static program analysis. ACM Computing Surveys 28, 340–343 (1996)
17. Midtgaard, J., Jensen, T.: A calculational approach to control-flow analysis by abstract interpretation. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 347–362. Springer, Heidelberg (2008)
18. Midtgaard, J., Jensen, T.P.: Control-flow analysis of function calls and returns by abstract interpretation. Information and Computation 211, 49–76 (2012)
19. Might, M., Manolios, P.: *A posteriori* soundness for non-deterministic abstract interpretations. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 260–274. Springer, Heidelberg (2009)
20. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology (TOSEM) 14(1) (January 2005)
21. Nielson, F., Nielson, H.R.: Interprocedural control flow analysis. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 20–39. Springer, Heidelberg (1999)
22. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: ACM Annual Conference (1972)
23. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(5) (August 2007)
24. Schmidt, D.A.: Natural-Semantics-Based abstract interpretation. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 1–18. Springer, Heidelberg (1995)
25. Schmidt, D.A.: Abstract interpretation of small-step semantics. In: Dam, M. (ed.) LOMAPS-WS 1996. LNCS, vol. 1192, pp. 76–99. Springer, Heidelberg (1997)
26. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, technical Report CMU-CS-91-145 (May 1991)
27. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: ACM SIGPLAN Symposium on Principles of Programming Languages, POPL (2011)
28. Van Horn, D., Might, M.: Abstracting abstract machines. In: ACM SIGPLAN International Conference on Functional programming, ICFP (2010)

Author Index