Lecture Notes in Computer Science 5505

Editorial Board

Stefan Kowalewski   Anna Philippou (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

15th International Conference, TACAS 2009
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2009
York, UK, March 22-29, 2009
Proceedings

Springer

Volume Editors

Stefan Kowalewski
RWTH Aachen, Embedded Software Laboratory
Ahornstr. 55, 52074, Aachen, Germany
E-mail: kowalewski@embedded.rwth-aachen.de

Anna Philippou
University of Cyprus, Department of Computer Science
1678 Nicosia, Cyprus
E-mail: annap@ucy.ac.cy

Foreword

ETAPS 2009 was the 12th instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 22 satellite workshops (ACCAT, ARSPA-WITS, Bytecode, COCV, COMPASS, FESCA, FInCo, FORMED, GaLoP, GT-VMT, HFL, LDTA, MBT, MLQA, OpenCert, PLACES, QAPL, RC, SafeCert, TAASN, TERMGRAPH, and WING), four tutorials, and seven invited lectures (excluding those that were specific to the satellite events). The five main conferences received 532 submissions (including 30 tool demonstration papers), 141 of which were accepted (10 tool demos), giving an overall acceptance rate of about 26%, with most of the conferences at around 25%. Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way of participating in this exciting event, and that you will all continue submitting to ETAPS and contributing towards making it the best conference on software science and engineering.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a confederation in which each event retains its own identity, with a separate Programme Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronised parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for 'unifying' talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2009 was organised by the University of York in cooperation with

▷ European Association for Theoretical Computer Science (EATCS)
▷ European Association for Programming Languages and Systems (EAPLS)
▷ European Association of Software Science and Technology (EASST)

and with support from ERCIM, Microsoft Research, Rolls-Royce, Transitive, and Yorkshire Forward.

The organising team comprised:

Chair	Gerald Luettgen
Secretariat	Ginny Wilson and Bob French
Finances	Alan Wood
Satellite Events	Jeremy Jacob and Simon O'Keefe
Publicity	Colin Runciman and Richard Paige
Website	Fiona Polack and Malihe Tabatabaie.

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Vladimiro Sassone (Southampton, Chair), Luca de Alfaro (Santa Cruz), Roberto Amadio (Paris), Giuseppe Castagna (Paris), Marsha Chechik (Toronto), Sophia Drossopoulou (London), Hartmut Ehrig (Berlin), Javier Esparza (Munich), Jose Fiadeiro (Leicester), Andrew Gordon (MSR Cambridge), Rajiv Gupta (Arizona), Chris Hankin (London), Laurie Hendren (McGill), Mike Hinchey (NASA Goddard), Paola Inverardi (L'Aquila), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Stefan Kowalewski (Aachen), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Gerald Luettgen (York), Rupak Majumdar (Los Angeles), Tiziana Margaria (Göttingen), Ugo Montanari (Pisa), Oege de Moor (Oxford), Luke Ong (Oxford), Catuscia Palamidessi (Paris), George Papadopoulos (Cyprus), Anna Philippou (Cyprus), David Rosenblum (London), Don Sannella (Edinburgh), João Saraiva (Minho), Michael Schwartzbach (Aarhus), Perdita Stevens (Edinburgh), Gabriel Taentzer (Marburg), Dániel Varró (Budapest), and Martin Wirsing (Munich).

I would like to express my sincere gratitude to all of these people and organisations, the Programme Committee Chairs and PC members of the ETAPS conferences, the organisers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the Organising Chair of ETAPS 2009, Gerald Luettgen, for arranging for us to hold ETAPS in the most beautiful city of York.

January 2009   Vladimiro Sassone, Chair
                                               ETAPS Steering Committee

Preface

This volume contains the proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009). TACAS 2009 took place in York, UK, 23–26 March, 2009, as part of the 12th European Joint Conferences on Theory and Practice of Software (ETAPS 2009), whose aims, organization, and history are presented in the foreword of this volume by the ETAPS Steering Committee Chair, Vladimiro Sassone.

TACAS is a forum for researchers, developers, and users interested in rigorously based tools and algorithms for the construction and analysis of systems. The conference serves to bridge the gaps between different communities that share common interests in tool development and its algorithmic foundations. The research areas covered by such communities include, but are not limited to, formal methods, software and hardware verification, static analysis, programming languages, software engineering, real-time systems, and communications protocols. The TACAS forum provides a venue for such communities at which common problems, heuristics, algorithms, data structures and methodologies can be discussed and explored. In doing so, TACAS aims to support researchers in their quest to improve the utility, reliability, flexibility, and efficiency of tools and algorithms for building systems.

The specific topics covered by the conference included but were not limited to: specification and verification techniques for finite and infinite-state systems; software and hardware verification; theorem-proving and model-checking; system construction and transformation techniques; static and run-time analysis; abstraction techniques for modeling and validation; compositional and refinement-based methodologies; testing and test-case generation; analytical techniques for secure, real-time, hybrid, critical, biological or dependable systems; integration of formal methods and static analysis in high-level hardware design or software environments; tool environments and tool architectures; SAT solvers; and applications and case studies.

TACAS traditionally considers two types of papers: research papers and tool demonstration papers. Research papers are full-length papers that contain novel research on topics within the scope of the TACAS conference and have a clear relevance for tool construction. Tool demonstration papers are shorter papers that give an overview of a particular tool and its applications or evaluation. TACAS 2009 received a total of 131 submissions including 22 tool demonstration papers and accepted 35 papers of which 8 papers were tool demonstration papers. Each submission was evaluated by at least three reviewers. After a six-week reviewing process, the program selection was carried out in a two-week electronic Program Committee meeting. We believe that the committee deliberations resulted in a strong technical program. The TACAS 2009 Program Committee selected Steven Miller (Rockwell Collins, USA) as an invited speaker, who kindly agreed

to give a talk entitled "Bridging the Gap Between Model-Based Development and Model Checking". The talk presented a translator framework that enables the use of several popular model checkers with commercial modeling tools and reported on its successful application in the development of avionics software. An abstract of this talk is included in this volume.

As TACAS 2009 Program Committee Co-chairs we would like to thank the authors of all submitted papers, the Program Committee members and all the referees for their invaluable contribution in guaranteeing such a strong technical program. We also thank Frank Holzwarth and Martin Karusseit for their prompt support with the Online Conference System used to manage the program selection process and Dominique Gückel for creating the TACAS 2009 webpage and helping with the preparation of the proceedings. Finally, we would like to express our appreciation to the ETAPS Steering Committee and especially its Chair, Vladimiro Sassone, as well as the Organizing Committee, chaired by Gerald Lüttgen, for their efforts in making ETAPS 2009 such a successful event.

January 2009   Stefan Kowalewski
                                                      Anna Philippou

Organization

Steering Committee

Ed Brinksma	ESI and University of Twente, The Netherlands
Rance Cleaveland	University of Maryland and Fraunhofer USA Inc., USA
Kim Larsen	Aalborg University, Denmark
Bernhard Steffen	University of Dortmund, Germany
Lenore Zuck	University of Illinois, USA

Program Committee

Marco Bernardo	University of Urbino, Italy
Ahmed Bouajjani	University of Paris 7, France
Ed Brinksma	ESI and University of Twente, The Netherlands
Alessandro Cimatti	FBK-IRST, Italy
Rance Cleaveland	University of Maryland and Fraunhofer USA Inc., USA
Swarat Chaudhuri	Pennsylvania State University, USA
Veronique Cortier	CNRS-LORIA, Nancy, France
Patrice Godefroid	Microsoft Research, Redmond, USA
Orna Grumberg	Technion, Israel Institute of Technology, Israel
Aarti Gupta	NEC Laboratories America Inc., USA
Nicolas Halbwachs	Verimag/CNRS, Grenoble, France
Michael Huth	Imperial College, UK
Kim Larsen	Aalborg University, Denmark
Stefan Kowalewski	RWTH Aachen, Germany
Thomas Kropf	Robert Bosch AG, Germany
Marta Kwiatkowska	University of Oxford, UK
Rupak Majumdar	University of California, Los Angeles, USA
Panagiotis Manolios	Northeastern University, USA
Radu Mateescu	INRIA/VASY, France
Ken McMillan	Cadence Berkeley Labs, USA
Anna Philippou	University of Cyprus, Cyprus
Andreas Podelski	University of Freiburg, Germany
C.R. Ramakrishnan	Stony Brook University, USA
Natasha Sharygina	University of Lugano, Switzerland
Oleg Sokolsky	University of Pennsylvania, USA
Bernhard Steffen	University of Dortmund, Germany
Frits Vaandrager	Nijmegen University, The Netherlands
Carsten Weise	RWTH Aachen, Germany
Lenore Zuck	University of Illinois, USA

Referees

Alessandro Aldini	Wolfgang Grieskamp	Petur Olsen
Christophe Alias	Alberto Griggio	Luke Ong
Rajeev Alur	Jan Friso Groote	Ghassan Oreiby
Eugene Asarin	Dominique Gückel	Ghassan Oreiby
Mohamed Faouzi Atig	Peter Habermehl	Rotem Oshman
Marco Bakera	Christine Hang	Luca Padovani
Sebastien Bardin	Faranak H. Dehkordi	Paritosh Pandya
Clark Barrett	Tamir Heyman	David Parker
Jasper Berendsen	Radu Iosif	Charles Pecheur
Nathalie Bertrand	Franjo Ivancic	Edgar Pek
Nikolaj Bjorner	Himanshu Jain	Knot Pipatsrisawat
Bernard Boigelot	Sven Jörges	Nir Piterman
Benedikt Bollig	Line Juhl	Lorenzo Platania
Edoardo Bontà	Yan Jurski	Vinayak Prabhu
Götz Botterweck	Vineet Kahlon	Polyvios Pratikakis
Bouyer Patricia	Joost-Pieter Katoen	Shaz Qadeer
Marco Bozzano	Mark Kattenbelt	Harald Raffelt
Aaron Bradley	Katya Kisyova	Sylvain Rampacek
Roberto Bruttomesso	Naoki Kobayashi	Arend Rensink
Véronique Bruyère	Piotr Kordy	Thomas Reps
Sebastian Burckhardt	Daniel Kroening	Pierre-Alain Reynier
Pavol Cerny	Shuvendu Lahiri	Noam Rinetzky
Krishnendu Chatterjee	Anna-Lena Lamprecht	Christophe Ringeissen
Vivien Chinnapongse	Frédéric Lang	Marco Roveri
Gianfranco Ciardo	Rom Langerak	Oliver Rüthing
Pedro R. D'Argenio	Etienne Lantreibecq	Theo Ruys
Alexandre David	Mikkel Larsen Pedersen	Vadim Ryvchin
Jed Davis	Jerome Leroux	Sriram
Leonardo de Moura	Shuhao Li	Sankaranarayanan
Stéphane Demri	Gavin Lowe	Bastian Schlich
Peter Dillinger	Maik Merten	John Schommer
Nikhil Dinesh	Andrea Micheli	Viktor Schuppan
Markus Doedt	Marius Mikucionis	Roberto Sebastiani
Susanna Donatelli	Ralf Mitsching	Olivier Serre
Laurent Doyen	David Monniaux	Wendelin Serwe
Constantin Enea	Sergio Mover	Sarai Sheinvald
Ansgar Fehnker	Andrzej Murawski	Sharon Shoham
Jeff Fischer	Ralf Nagel	Mihaela Sighireanu
Pascal Fontaine	Wonhong Nam	Nishant Sinha
Anders Franzen	K. Narayan Kumar	Jeremy Sproston
Pierre Ganty	Johannes Neubauer	Sudarshan Srinivasan
Hubert Garavel	Thomas Noll	Jan Stoecker
Amit Goel	Gethin Norman	Andrei Tchaltsev
Marco Gribaudo	Ulrik Nyman	Claus Thrane

Table of Contents

Model Checking I

Tools I

Game-Theoretic Approaches

Verification of Concurrent Programs

Tools II

Model Checking II

Parametric Analysis

Generative Approaches

Program Analysis

Hybrid Systems

Decision Procedures and Theorem Proving

Invited Contribution

Hierarchical Set Decision Diagrams and Regular Models[⋆]

Yann Thierry-Mieg, Denis Poitrenaud, Alexandre Hamez, and Fabrice Kordon

Université P. & M. Curie, LIP6 - CNRS UMR 7606 - 4 Place Jussieu, Paris, France
`first.last@lip6.fr`

Abstract. This paper presents algorithms and data structures that exploit a compositional and hierarchical specification to enable more efficient symbolic model-checking. We encode the state space and transition relation using hierarchical Set Decision Diagrams (SDD) [9]. In SDD, arcs of the structure are labeled with sets, themselves stored as SDD.

To exploit the hierarchy of SDD, a structured model representation is needed. We thus introduce a formalism integrating a simple notion of *type* and *instance*. Complex composite behaviors are obtained using a synchronization mechanism borrowed from process calculi. Using this relatively general framework, we investigate how to capture similarities in regular and concurrent models. Experimental results are presented, showing that this approach can outperform in time and memory previous work in this area.

1 Introduction

Model checking is a formal verification approach that suffers from the state-space explosion problem. One approach which has been successfully used to tackle this problem is symbolic model-checking using binary decision diagrams [3].

Shared reduced ordered Binary Decision Diagrams (Binary DD or BDD) offer in many cases a very compact representation of a binary function of n Boolean variables, i.e. a function $\mathbb{B}^n \mapsto \mathbb{B}$. BDD rely on a unique table to avoid creating nodes more than once: the decision tree is built from the leaves (the terminals 0 and 1) up to the unique root. This yields a canonical representation for a boolean function given an ordering of its variables. Thus comparison of two BDD is of constant complexity. Using a cache, it is possible to obtain algorithms in complexity polynomial to the number of nodes in the data structure, rather than to the number of paths. For instance, union (or) and intersection (and) of two BDD a and b has a complexity proportional to the product of the number of nodes in the representation a and b.

Since their introduction, many extensions to BDD have been proposed. One family of extensions consists in Multi-Terminal DD (MTBDD [5]) a.k.a. Algebraic DD [1] which allow to represent functions $\mathbb{B}^n \mapsto \mathbb{N}$, and by extension when the set of terminals remains of manageable size $\mathbb{B}^n \mapsto \mathbb{R}$. This type of DD has been successfully used for probabilistic model-checking [7], as well as being competitive with sparse representations for matrix computations [1].

Another family of extensions is Multiway DD [5], or Data DD [8] which allow to store functions $\mathbb{N}^n \mapsto \mathbb{B}$, or even $\mathbb{N}^n \mapsto \mathbb{N}$ when combining with multi-terminals. Although comparable to binary encodings when variables are bounded, they allow to handle *a priori* unbounded variables, and may provide more efficient solutions.

Finally, many dedicated data structures that use the same basic concepts of canonical representation and dynamic programming have emerged to tackle timed (e.g. Clock Difference Diagrams CDD [2], Clock Region Diagrams CRD [12]) or probabilistic systems (e.g. MatriX Decision Diagrams MxD [7]).

In hierarchical Set Decision Diagrams (SDD [9]) arcs are labeled by a set of values rather than a single valuation. They represent assignment sequences of the form $\omega_1 \in s_1; \cdots; \omega_n \in s_n$ where ω_i are variables and s_i are sets of values. Since sets are compactly represented using decision diagrams, the arcs of the structure may be labeled by SDD (or indeed any other variant of DD), introducing hierarchy in the data structure. This produces a fundamental difference with other decision diagram types by allowing similar subsystems of a larger specification to share their representation. In the case of very regular systems, they may even provide an exponential compression factor with respect to usual DD [11].

Contributions: In this paper we investigate how SDD can be used to provide an efficient representation of the state space of composite systems. We define a general framework to express systems as a composition of smaller (possibly similar) subsystems using a notion of *type* and *instance*. Subproblems are composed using an event-based synchronization model borrowed from process calculi.

We investigate *when* gains from increased sharing can be expected from using SDD and how to maximize the gain when applicable. For a standard benchmark set of parametric models borrowed from [5] we show that the SDD solution is more efficient in both time and memory than the current state of the art in symbolic representations.

Outline: Section 2 defines SDD and formalizes operations over SDD as inductive homomorphisms. Section 3 defines a general formalism that allows to closely match the requirements of SDD based solutions. Section 4 then investigates diverse ways of encoding a problem. Finally, section 5 compares the different encodings proposed across a benchmark of models.

2   Context

This section recalls the salient points of Hierarchical Set Decision Diagrams, a data structure based on the principles of decision diagram technology (node uniqueness thanks to a canonical representation, dynamic programming, ordering issues . . .). They feature two main original aspects: the support of hierarchy in the representation (section 2.1) and the definition of user operations through a mechanism called *inductive homomorphisms* (section 2.2) which gives freedom and flexibility to the user. Usually, the next state function of a system is encoded using one or more decision diagrams, with two variables per variable of the state signature.

2.1   Set Decision Diagrams

Hierarchical Set Decision Diagrams (SDD) defined in [9], are shared decision diagrams in which arcs are labeled by a *set* of values, instead of a single value. This set may itself

be represented by an SDD, thus when labels are SDD, we think of them as hierarchical decision diagrams. Definition 1 is taken practically verbatim from [11] where it was adapted for more clarity from [9].

SDD are data structures for representing sets of sequences of assignments of the form $\omega_1 \in s_1; \omega_2 \in s_2; \cdots; \omega_n \in s_n$ where ω_i are variables and s_i are sets of values.

We assume no variable ordering, and the same variable can occur several times in an assignment sequence. We define the terminal 1 to represent the empty assignment sequence, that terminates any valid sequence. The terminal 0 represents the empty set of assignment sequences. In the following, *Var* denotes a set of variables, and for any ω in *Var*, $\mathrm{Dom}(\omega)$ represents the domain of ω *which may be infinite*.

Definition 1 (Set Decision Diagram). $\delta \in \mathbb{S}$, *the set of SDD, is inductively defined by:*

- $\delta \in \{0, 1\}$ *or*
- $\delta = \langle \omega, \pi, \alpha \rangle$ *with:*
  - $\omega \in Var$.
  - $\pi = s_0 \cup \cdots \cup s_n$ *is a finite partition of* $\mathrm{Dom}(\omega)$, *i.e.* $\forall i \neq j, s_i \cap s_j = \emptyset, s_i \neq \emptyset, n$ *finite.*
  - $\alpha : \pi \to \mathbb{S}$, *such that* $\forall i \neq j, \alpha(s_i) \neq \alpha(s_j)$.

By convention, when it exists, the element of the partition π *that maps to the SDD* 0 *is not represented.*

Despite its simplicity, this definition supports rich and complex data:

- SDD support domains of infinite size (e.g. $\mathrm{Dom}(\omega) = \mathbb{R}$), provided that the partition size remains finite (e.g. $]0..3],]3.. + \infty]$). This feature could be used to model clocks for instance (as in [12]). It also places the expressive power of SDD above most variants of DD.
- SDD or other variants of decision diagrams can be used as the domain of variables, introducing hierarchy in the data structure.
- SDD can handle paths of variable lengths, if care is taken when choosing the state encoding to avoid creating so-called incompatible sequences (see [9]). This feature is useful when representing dynamic structures such as queues, lists or variable size arrays.

2.2   Operations and Homomorphisms

SDD support standard set theoretic operations $(\cup, \cap, \setminus)$. They also offer a concatenation operation $\delta_1 \cdot \delta_2$ which replaces 1 terminal of δ_1 by δ_2. This corresponds to a cartesian product. In addition, basic and inductive homomorphisms are introduced as a powerful and flexible mechanism to define application specific operations. A detailed description of homomorphisms including many examples can be found in [8].

A basic homomorphism is a mapping $\Phi : \mathbb{S} \mapsto \mathbb{S}$ satisfying $\Phi(0) = 0$ and $\forall \delta, \delta' \in \mathbb{S}, \Phi(\delta + \delta') = \Phi(\delta) + \Phi(\delta')$. The sum $+$ and the composition $\circ$ of two homomorphisms are homomorphisms. For instance, the homomorphism $\delta \cdot Id$, where $\delta \in \mathbb{S}$ and *Id* designates the identity homomorphism, permits to left concatenate sequences. We widely use the left concatenation of a single assignment $(\omega \in s)$, noted $\omega \xrightarrow{s} Id$. Many basic homomorphisms are hard-coded.

Furthermore, application-specific mappings can be defined by *inductive* homomorphisms. An inductive homomorphism ϕ is defined by its evaluation on the 1 terminal $\phi(1) \in \mathbb{S}$, and its evaluation $\Phi' = \phi(\omega, s)$ for any $\omega \in Var$ and any $s \subseteq \text{Dom}(\omega)$. The expression $\phi(\omega, s)$ is itself a (possibly inductive) homomorphism, that will be applied on the successor node $\alpha(s)$. The result of $\phi(\langle \omega, \pi, \alpha \rangle)$ is then defined as $\sum_{s \in \pi} \phi(\omega, s)(\alpha(s))$, where $\sum$ represents a union.

As an example, the local construction $\mathcal{L}$ allows to "carry" a homomorphism h to a certain variable v, and apply h to the current state of v. Thus, it implements an operation local to the variable v. This homomorphism will be used in section 3. It is defined by:

$$\mathcal{L}(v, h)(\omega, s) = \begin{cases} \omega \xrightarrow{s} \mathcal{L}(v, h) \ if \omega \neq v \\ \omega \xrightarrow{h(s)} Id \quad else \end{cases} \quad \mathcal{L}(v, h)(1) = 0$$

The **transitive closure** * unary operator allows to perform a least fixpoint computation. For any homomorphism h and any node $\delta \in \mathbb{S}$, $h^\star(\delta)$ is evaluated by repeating $\delta \leftarrow h(\delta)$ until a fixpoint is reached. In other words, $h^\star(\delta) = h^n(\delta)$ where n is the smallest integer such that $h^n(\delta) = h^{n+1}(\delta)$. This operator is often applied to $(Id + h)$ instead of just h, allowing to accumulate newly computed assignment sequences in the result.

An important recent result is that we have defined a set of rewriting rules for homomorphisms [11], allowing to automatically make use of the decision diagram saturation algorithms originally due to Ciardo [6]. When computing the least fixpoint of a transition relation over a set of states, this algorithm offers gains of one to three orders of magnitude over classical BFS fixpoint algorithms.

For the user, these rewriting rules are transparent. Given a set of homomorphisms $\{t_1, \dots, t_n\}$ that represent a partition of the transition relation of the system, the application of $(t_1 + \dots + t_n + Id)^\star$ to a node automatically triggers the saturation algorithm for the evaluation. Note that this is a central operation in any symbolic model-checking problem since reachability is defined as a transitive closure over the full transition relation. Futhermore a more complex CTL model-checker can then be constructed using nested transitive closures over the transition relation or its reverse [4].

3   Instantiable Transition System

This section introduces a framework to define formalisms in a way that allows to take advantage of the characteristics of SDD. Previous manual encoding of some particular systems [11] has shown that a best case exponential compression factor can be reached by SDD with respect to other DD. To generalize these results, we define Instantiable Transition System (ITS), a minimal Labeled Transition System (LTS) style formalism that makes use of the notions of *type* and *instance* to emphasize locality of actions. This helps identify similar subproblems. The requirements we express through this formalism on the input language are sufficiently wide to encompass many types of description languages. Any formalism that can fit this generic description is likely to gain from using SDD rather than other "flat" decision diagram types.

3.1   ITS Definition

Notations: $\text{Bag}(A)$ denotes a multiset over a set A. Let $\oplus$ be a commutative operation $A \times A \mapsto A$. Let $\tau \in \text{Bag}(A)$, we note $S = \bigoplus_{a \in \tau} a$ where if an element $a \in A$ occurs n

times in τ it will be $\oplus$-ed n times in S. We note $tuple.X, tuple.Y \ldots$ the element X (resp. $Y \ldots$) of a tuple $tuple = \langle X, Y, \cdots \rangle$.

The generic definition of an Instantiable Transition System (ITS) builds upon the notion of model type and instance. It uses a composition mechanism based solely on transition *synchronization* (no explicit shared memory or channel). Definition 2 sets an abstract contract or interface that must be realized by concrete ITS types. The principle is to build hierarchical models in which elementary bricks are homogeneous to composite models, as they both conform to the notion of ITS type.

Definition 2 (ITS Concepts). *An **ITS type** must provide a tuple* $type = \langle S, InitStates, T, Locals, Succ \rangle$:

- *S is a set of states;*
- *InitStates $\subseteq$ S is a finite subset of designated initial states;*
- *T is a finite set of public transition labels;*
- *Locals : $S \mapsto 2^S$ is the local successors function.*
- *Succ : $S \times \mathrm{Bag}(T) \mapsto 2^S$ is the transition function satisfying $\forall s \in S, Succ(s, \emptyset) = \{s\}$.*

*Let Types denote a set of ITS types. An **ITS instance** i is defined by its ITS type, noted $type(i) \in Types$. An ITS instance i may be associated to a state $s \in type(i).S$. We use the terminology: "assign a state s to instance i".*

InitStates is introduced to avoid violating encapsulation: to initialize an instance we need to be able to designate its initial configuration(s) without knowing the internal structure of the instance.

Locals will typically return states reachable through occurrence of local events. It represents transitions that may occur within an instance autonomously or independently from the rest of the system.

The function *Succ* allows to obtain successors by explicitly synchronizing over a multiset of public transition labels. Synchronizing on an empty multiset of transitions leaves the state of the instance locally unchanged. *Succ* takes a multiset of transition labels as argument, to resolve ambiguities that may occur when synchronizing several labels of a given instance The definition of the *Succ* as returning a set of successors (and not a single successor state) offers good generality, and allows in particular to capture non-deterministic transition relations as we will show in section 3.4. This feature allows a compact transition relation representation, as shown by the example of section 4.

Note that *Succ* is the only way to control the behavior of a (sub)system from outside. Thus the transition relation of a full system can only be defined in terms of transition synchronizations using *Succ* and of independent local behaviors. The definition of composition as a synchronization of independent effects on parts of a system (rather than data or channel sharing) is favorable to using various verification algorithms that exploit compositional verification [5,10] and locality of actions.

These functions will be used to define the semantics of a composite type below. To encode a system using SDD, one must define an SDD encoding of a state $s \in S$, and a homomorphism encoding of each of the two functions *Locals* and *Succ*.

As an example, consider the graphical type declaration of a process and buffer type depicted in Fig. 1. This example taken from [5] describes a round robin protocol allowing to share a single buffer to communicate among n processes. The buffer will initially
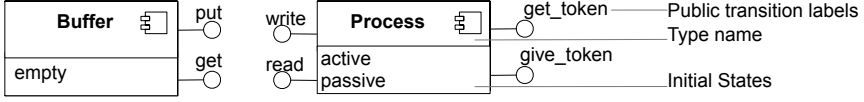
Fig. 1. Two type declarations for a resource and a process. Encapsulation makes implementation details irrelevant: only public transition labels (interface) and initial states are visible. The processes (instances of the Process type) will be connected through "get_token" and "give_token" to form a ring topology. The processes "get" ("read") the message that was "put" ("write") in the buffer by another process.

be *empty*. Initially, a single process will be *active* (has the token), all others will be *passive*. This round robin model will be used as a running example through the rest of the paper.

A full system is defined by an instance of a particular type in a specific initial state. As a full system is self-contained, the definition of reachability only depends on the definition of *Locals*:

(Reachability). A state s' is reachable by an instance i from the state s_0 iff. $\exists s_1, \ldots s_n \in type(i).S$ s.t. $s' = s_n \wedge \forall 1 \leq i \leq n, s_i \in type(i).Locals(s_{i-1})$.

3.2 A Composite Type

We now define a composite ITS type to offer support for the hierarchical composition of ITS instances.

Notations: Let I designate a set of ITS instances. CS_I designates the set of functions that map instances $i \in I$ to a state of $type(i)$, i.e. $cs \in CS_I \implies \forall i \in I, cs(i) \in type(i).S$. $Syncs_I$ designates the set of functions that map instances $i \in I$ to a multiset of public transition labels of $type(i)$, i.e. $t \in Syncs_I \implies \forall i \in I, t(i) \in \text{Bag}(type(i).T)$. The sum $\oplus : Syncs_I \times Syncs_I \mapsto Syncs_I$ is defined as: $t = t_0 + t_1 \iff \forall i \in I, t(i) = t_0(i) + t_1(i)$ where $+$ designates the standard sum of multisets.

Intuitively, CS_I represents composite states, an element of $Syncs_I$ corresponds to a synchronization of public labels of the set I of subcomponents. The sum $\oplus$ represents an operation cumulating the effect of two synchronizations. For instance, let $I = \{i_0, i_1\}$. Let $s_0, s_1 \in Syncs_I$, $s_0(i_0) = t_0 + 2't_1$, $s_0(i_1) = \emptyset$; $s_1(i_0) = t_0$, $s_1(i_1) = t_3$. Then $s_2 = s_0 \oplus s_1 \implies s_2(i_0) = 2't_0 + 2't_1$, $s_2(i_1) = t_3$.

We define the next state function $Next_I$, which is used when defining *Locals* and *Succ* below $Next_I : CS_I \times \text{Bag}(Syncs_I) \mapsto 2^{CS_I}. \forall s, s' \in CS_I, \forall \tau \in \text{Bag}(Syncs_I)$,

$$s' \in Next_I(s, \tau) \text{iff} \forall i \in I, s'(i) \in type(i).Succ(s(i), (\bigoplus_{t \in \tau} t)(i)).$$

Definition 3 (Composite). *A **composite** is a tuple $C = \langle I, IS, ST, V \rangle$:*

- *I is a finite set of ITS instances, said to* be contained *by C. We further require that the type of each ITS instance preexists when defining these instances, in order to prevent circular or recursive type definitions.*
- *$IS \subseteq \{s \in CS_I \mid \forall i \in I, s(i) \in type(i).InitStates\}$ is a finite set of designated initial states*

- $ST \subset Syncs_I$ is the finite set of synchronizations;
- $V : ST \mapsto \{public, private\}$ assigns a visibility to each synchronization

The ITS type corresponding to a composite, is defined as:

- $S = CS_I$
- $InitStates = IS$
- $T = \{st \in ST \mid V(st) = public\}$
- $Locals : S \mapsto 2^S.\ \forall s, s' \in S, s' \in Locals(s)$ iff

$$\exists i \in I, s'(i) \in type(i).Locals(s(i)) \wedge \forall j \in I, j \neq i, s'(j) = s(j)$$
$$or\ \exists t \in ST, V(t) = private, s' \in Next_{C.I}(s, \{t\})$$

- $Succ : S \times \mathrm{Bag}(T) \mapsto 2^S.\ \forall s, s' \in S, \forall \tau \in \mathrm{Bag}(T), Succ(s, \tau) = Next_{C.I}(s, \tau)$

Definition 3 is a realization of the generic ITS type contract. It contains either elementary subcomponents (see section 3.3), or recursively other instances of composite nature.

Locals is defined as states reachable through the occurrence of local transitions of any nested component (without affecting the other subcomponents) or states reachable through occurrence of any given private synchronization.

Succ is realized by "summing" the impact of the multiset of transitions given as its argument using the $\oplus$ operator defined over $Syncs_I$, and synchronously updating the state of each subcomponent.

As an example consider in Fig. 2 a composite type built to represent the round robin system with two processes.
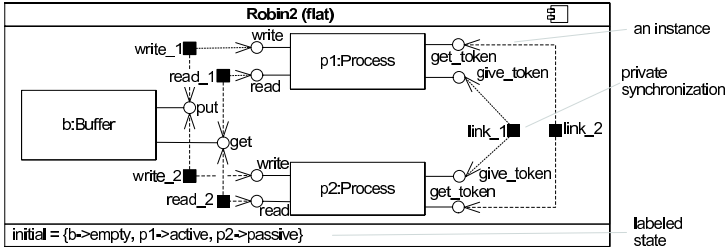
Fig. 2. A composite type declaration, containing three instances and six private synchronizations. For instance, private synchronization $link_2$ is read $link_2(p1) = get\_token, link_2(p2) = give\_token, link_2(b) = \emptyset$.

Encoding: A state $s \in CS_I$ of a composite C will be represented by an SDD of $|I|$ variables, each representing the state of an instance $i \in I$. The domain of each variable is determined by the type of the instance. The $Next_I$ function is defined using the $\mathcal{L}$ homomorphism introduced in section 2.2. For any $\tau \in \mathrm{Bag}(T)$:

$$Next_I(\tau) = \bigcirc_{i \in I} \mathcal{L}(i, type(i).Succ((\bigoplus_{t \in \tau} t)(i)))$$

The homomorphisms representing *Locals* and *Succ* , $\forall \tau \in \mathrm{Bag}T$, are encoded:

$Locals\ \ = \sum_{i \in I} \mathcal{L}(i, type(i).Locals) + \sum_{t \in ST, V(t) = private} Next_{C.I}(\{t\})$
$Succ(\tau) = Next_{C.I}(\tau)$

3.3   An Elementary Type

To have a fully working model definition, we still need to define an elementary type. We use here a labeled transition system as elementary brick, adapted to the ITS type contract. In practice any finite state model is appropriate.

Definition 4. *An elementary Labeled Transition System (LTS) is a tuple*
$LTS = \langle N, N_0, L, E, V \rangle$:

- N *is a finite set of nodes;*
- $N_0 \subseteq N$ *is a subset of designated initial states;*
- L *is a finite set of labels for transitions;*
- $E \subseteq N \times L \times N$ *is a set of labeled edges;*
- $V : L \mapsto \{public,\ private\}$ *is a function assigning a* visibility *to each label;*

The ITS type which corresponds to an elementary LTS, is defined as:

- $S = N$
- $InitStates = N_0$
- $T = \{l \in L \mid V(l) = public\}$
- $Locals : S \mapsto 2^S$ *is defined as* $n' \in Locals(n)$ iff $\exists l \in L, V(l) = private \wedge \langle n, l, n' \rangle \in E$;
- $Succ : S \times Bag(T) \mapsto 2^S$: $Succ(n, \tau) = \emptyset$ if τ contains more than one transition label. Else, when $\tau = \{l\}$, $\forall n, n' \in S, n' \in Succ(n, \{l\})$ iff $\langle n, l, n' \rangle \in E$.

As an example, Fig. 3 represents an implementation of the Process type introduced earlier (Fig. 1) using an LTS.
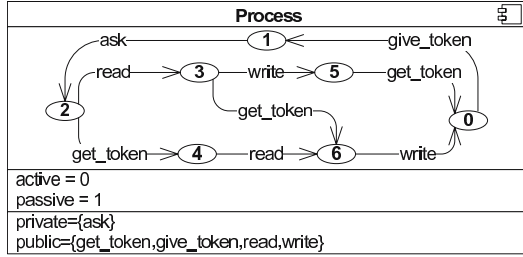
Fig. 3. An LTS representing an implementation of the Process type in the round robin protocol

Encoding: we index the states of LTS, and use an SDD with a single variable of integer domain reflecting the current state. The transition relation is easily realized using a precomputed transition function $f : S \times L \mapsto 2^S$ such that $f(n, l) = \{n' \mid \langle n, l, n' \rangle \in E\}$. Furthermore, for any set $s \subseteq S$ we define the set $priv(s) = \bigcup_{x \in s} \bigcup_{l \in L \wedge V(l) = private} f(x, l)$.

Locals and *Succ* are defined using inductive homomorphisms. Note that as we have a single variable in the encoding these homomorphisms do not need to propagate. *Succ* is defined below when the argument $\tau \in Bag(T)$ contains a single transition label l. Otherwise it returns the terminal 0.

$$\begin{cases} Locals(\omega, s) = e \xrightarrow{priv(s)} Id \\ Locals(1) \quad = 1 \end{cases} \begin{cases} Succ(l)(\omega, s) = e \xrightarrow{\bigcup_{x \in s} f(x, l)} Id \\ Succ(l)(1) \quad = 1 \end{cases}$$

3.4 An Extended Composite Type

The concepts introduced up to this point offer basic support for the definition of hierarchical models. Extending this definition is possible provided that the ITS type contract is preserved. We consider here such an extension, which proposes an additional type of synchronization to handle non-determinism.

This additional construct allows more compact modeling, and a more efficient encoding of the transition relation. An example using this extended composite definition and illustrating its benefits is presented in section 4.

Notations: We define an additional operator to combine $Syncs_I$. Let the product $\otimes$: $2^{Syncs_I} \times 2^{Syncs_I} \mapsto 2^{Syncs_I}$ be defined as:

$$\forall A, B \subseteq Syncs_I, A \otimes B = \{a \oplus b \mid a \in A \land b \in B\}$$

We then define a slightly more complex composite type :

Definition 5. *A **non-deterministic composite** is a tuple $NDC = \langle C, K \rangle$:*

- *C is a composite type*
- *$K : C.T \mapsto \{AND, XOR\}$ partitions transitions into basic AND kind synchronizations and non deterministic choice XOR kind synchronizations;*

We define the determinize function $Det : \mathrm{Bag}(T) \mapsto 2^{Syncs_I}$:

$$Det(\tau) = \begin{cases} \{\bigoplus_{\{t \in \tau | K(t) = AND\}} t\} \otimes \bigotimes_{\{t \in \tau | K(t) = XOR\}} t & \text{if } \exists t \in \tau, K(t) = XOR \\ \{\bigoplus_{\{t \in \tau | K(t) = AND\}} t\} & \text{otherwise} \end{cases}$$

Det allows to map the semantics of XOR synchronizations to a set of deterministic AND synchronizations. To realize the ITS type definition, we define:

- *$S = C.S$, InitStates $= C.InitStates$, $T = C.T$*
- *$Locals : S \mapsto 2^S$. $\forall s, s' \in S, s' \in Locals(s)$ iff*

$$\begin{cases} \exists i \in C.I, s'(i) \in type(i).Locals(s(i)) \land \forall j \in C.I, j \neq i, s'(j) = s(j) \\ or \ \exists \theta \in C.ST, C.V(\theta) = private, \exists t \in Det(\theta), s' \in Next_{C.I}(s, \{t\}) \end{cases}$$

- *$Succ : S \times \mathrm{Bag}(T) \mapsto 2^S$. $\forall s, s' \in S, \forall \sigma \in \mathrm{Bag}(T)$, $s' \in Succ(s, \sigma)$ iff $\exists \tau \in Det(\sigma), s' \in Next_{C.I}(s, \tau)$.*

Note that an extended composite in which no disjunctive synchronizations are defined is identical to definition 3. However, the extended definition introduces "exclusive or" type synchronizations, in which only one of the transition labels that belong to the set $Syncs_I$ is required to occur when the synchronization occurs. The transition label is chosen arbitrarily. The function Det selects one transition label from each XOR synchronization, and all transition labels from the AND transitions. Its output is a set of $Syncs_I$ that can be used to define a Successors rule using the same $Next$ function as definition 3.

Encoding: The state encoding is the same as the encoding for a (basic) composite. The homomorphisms representing *Locals* and *Succ* , $\forall \sigma \in \mathrm{Bag}T$, are encoded:

$$Locals = \sum_{i \in I} \mathcal{L}(i, type(i).Locals) + \sum_{\tau \in ST, V(\tau) = private} (\sum_{t \in Det(\tau)} Next_{C.I}(\{t\}))$$
$$Succ(\sigma) = \sum_{\tau \in Det(\sigma)} Next_{C.I}(\tau)$$

4   Hierarchical Modeling Strategies

ITS allow to model a given system in a number of equivalent ways, depending on the hierarchy of types that is defined. One way of seeing this is that ITS offer to *parenthesize* a parallel composition of *n* processes. Flattening the representation is always possible, yielding an equivalent composite ITS containing only instances of an elementary type. This can be seen as removing parenthesis from the expression of the synchronization, which does not affect the resulting semantics. However, a model's hierarchy allows to factorize description of similar structures and behaviors. This is exploited to provide a more efficient SDD solution for model-checking.

For instance, to obtain an homogeneous representation of a chain of processes and a single process, we can define a type *ProcessGroup* (Fig. 4) to contain a set of process instances. This homogeneous representation is only possible thanks to the use of XOR synchronizations; a simpler encoding using only AND synchronizations would require that a process group containing *n* process instances make visible *n* versions of the *write* and *read* transitions.

The process group is then identical to a process, from the ITS point of view that sees only public transition labels and designated initial states. We will note such a composite $M2 = (P \parallel P)$, where *P* represents an elementary process type, and $\parallel$ denotes a parallel composition. This homogeneous representation of a set of processes allows to build a larger process group by combining two process groups using the same schema. For instance, we can define $M4 = (M2 \parallel M2)$ using two instances of the type defined in Fig. 4 to represent 4 process.
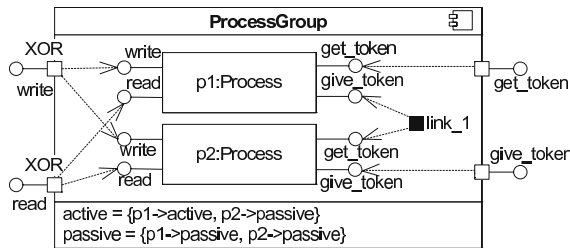
Fig. 4. An extended composite ITS representing a group of Process in a manner homogeneous to a single Process. This pattern can be generalized to contain *k* instances rather than just two.

We can then define (Fig. 5) a *ProcessRing* type that models the boundary synchronizations necessary to close a process ring.

We compare here two approaches to encode a system of *n* process. The Recursive(*grain*) approach consists in building process groups such that no process group definition ever contains more than *grain* process group instances.

The Group(*grain*) approach consists in building a process group type containing $n/grain$ subgroups of sizes ranging from *grain* to $grain + 1$. The subgroups are defined in this approach as a simple composition of *grain* (or $grain + 1$) elementary Process instances. The overall depth is thus two: the Process group composite contains subgroup composite instances that contain elementary type instances.
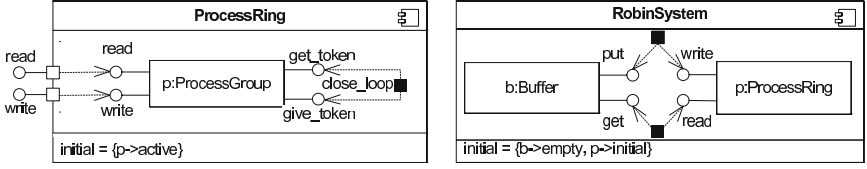
Fig. 5. A composite ITS representing the actions closing a Process ring and a full system as a composite of a Buffer and a ProcessRing

For instance, let us compare for $n = 18$ the encodings of Robin(18). Recursive(2) would yield $M2 = (P \parallel P), M4 = (M2 \parallel M2), M8 = (M4 \parallel M4), M16 = (M8 \parallel M8)$, and finally $M18 = (M16 \parallel M2)$. Recursive(3) would yield $M3 = (P \parallel P \parallel P), M9 = (M3 \parallel M3 \parallel M3), M18 = (M9 \parallel M9)$ with more shallow hierarchy. Group(4) would build a model $M4 = (P \parallel P \parallel P \parallel P), M5 = (P \parallel P \parallel P \parallel P \parallel P), M18 = (M5 \parallel M5 \parallel M4 \parallel M4)$.

Comparing Strategies. The Recursive approach fixes the maximum number of variables in an SDD assignment sequence, and lets "depth" in the hierarchy grow with $log_{grain}(n)$. The Group approach fixes the depth at 2 and fixes the length of assignment sequences in the "deeper" level to *grain*. The number of variables in the outer level thus grows with $n/grain$.

We have run experiments with these strategies, for three examples taken from Smart benchmarks [5]. **Robin** is the protocol we have used as running example in this paper. **Philosophers** models Dijkstra's classical dining philosophers. **Slotted Ring** describes a ring communication protocol with one slot per participant. These three models are regular, i.e. parametric in the number of participants in the protocol. They are thus all appropriate to apply our encoding strategies.

We report performance for Robin using the XOR construct. Without it, locality of events which is critical to saturation performance is broken. As a result, even if the final state space representation size is the same, the poorer encoding of the transition relation yields high degree polynomial complexity in n due to peaks in representation size.

Table 1 presents the results obtained with two of these regular models. In this experiment we let the grain vary in both Recursive and Group approaches.

Our experiments shows on Robin and Philosophers that the Recursive strategy can be extremely efficient w.r.t. to the Group strategy.

The results on Philosophers have been omitted due to space constraints, but performance is sublinear $O(n)$ in the Group approach (with larger grain yielding better performance) and logarithmic $O(ln(n))$ in recursive approaches.

For Robin, the final state space representation size is $O(ln(n))$, like Philosophers. However, the assymetry of the initial state enforces n iterations to cover all positions of the token in the ring, yielding overall linear complexity in n.

However, the recursive encoding of the model description, even when it is possible (i.e. the model is regular) does not ensure that the state-space computation will be easy, as the Ring example shows. In this model the recursive approach does not yield a final representation size in $O(ln(n))$. Although the system is regular, system-wide dependencies between component states force a larger representation, in which most arcs bear a single value. The increased depth in the data structure even introduces overhead in this case, thus the Group approach is more effective.

Table 1. Compared performances of Recursive and Group approaches for a sampling of *grain* parameter. "–" entries indicate failure due to exhaustion of memory.

		Recursive				Group					
grain ⇒		3		5		1		5		10	
Model Size	States #	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)
Slotted Ring											
50	1.7×10^{52}	4.9	59.7	2.2	48.1	2.5	112.7	1.9	55.9	2.5	50.2
100	2.6×10^{105}	94.7	463.9	27.9	300.1	19.7	814.4	14.5	410.5	17.6	342.6
200	8.4×10^{211}	-	-	489.9	2285.2	-	-	-	-	128.7	2735.7
Robin											
100	2.8×10^{32}	0.24	12.5	0.26	11.8	26.9	102.8	1.0	23.8	0.7	14.8
400	2.3×10^{123}	1.0	45.4	1.2	43.8	998.2	1118.7	62.9	318.5	15.5	169.2
1000	2.4×10^{304}	3.0	117.8	3.12	106.6	*> 1000*	*> 2473.2*	*> 1000*	*> 1718.1*	307.7	1006.5

The setting Group(1) closely mimics a flattened composition of processes of the form $(P \mathbin{/\!/} P \mathbin{/\!/} \cdots \mathbin{/\!/} P)$. Since this is the encoding other DD based tools would use (no hierarchy), it is a good baseline comparison.

We can observe that using larger variable domains (i.e. increasing the *grain*) tends to reduce complexity in both approaches. This trend is reversed when the grain is so large that the depth is very shallow in *Recursive*, or the outer assignment sequence is too short in *Group*.

5   Comparative Performance Analysis

This section presents performance comparisons of our tool that relies on ITS and SDD to the tool Smart based on MDD [5]. To allow us to easily use Smart's benchmark models, our tool uses place transition nets as an elementary type rather than LTS.

Comparison to Smart is indicated as it is, to our knowledge, the only other symbolic model-checker that uses a saturation algorithm. Comparisons to NuSMV were also performed, but are not really comparable, as without saturation it cannot compete. In fact, no answer in a reasonable time was given for the parameters we use in our benchmark. This confirms the experimentations presented in [5].

Figure 6 presents the comparisons run for five parametric models taken from the Smart's own benchmark. These models were chosen as being representative of both tools behavior, with extreme cases represented by Philo (SDD more efficient) and Ring (MDD more efficient). Three of them are the regular models introduced in section 4. In those models, the number of variables increases with n, while variable domains are fixed and typically small (less than 20 values). We also included two parametric models which are not regular, and could not be encoded using the approaches of the previous section. **FMS** and **Kanban** model flexible manufacturing systems. Parameter n defines the number of available resources rather than the size of the manufacturing plant. In these models the number of variables is fixed, while variable domains evolve with n.

Both tools use the best available settings, and compute the full reachable state-space. The state space size is in all cases exponential in the parameter n.

Philosophers: the *Recursive* approach (see section 4) is so successful that time and memory are still negligible for the value $n = 1000$. The complexity is in $O(n)$ in Smart,
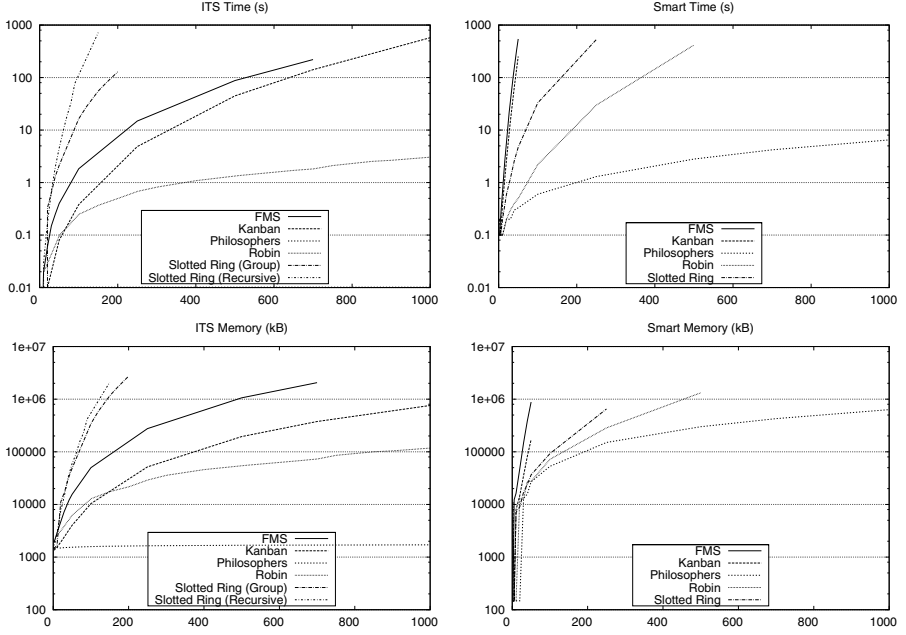
Fig. 6. Compared performances of ITS/SDD (left) vs. Smart (right), using the best settings for each tool. The *x* axis is the parameter *n* setting model's complexity. The logarithmic *y* axis represents time in seconds (top) and memory in kilo-bytes (bottom).

thanks to saturation, but it is $O(ln(n))$ with SDD. For instance, we compute reachable states of the Philosophers for $n = 10^{3000}$ in 36 seconds using 386 MB of RAM.

Robin: the complexity in Smart is high degree polynomial, where our *Recursive* solution is $O(n)$.

Ring: We report here both the results of the Recursive strategy (with grain=3) and of the Group strategy (with grain=10). Both strategies remain in high degree polynomial complexity comparable to Smart.

In this model, many arcs carry a single value even when using SDD. Since the resulting tree structure is similar, the theoretical complexity of both solutions is comparable. However, the lower memory footprint per node of MDD vs SDD factors up to give Smart the advantage. It was able to compute *Ring* up to $n = 250$ when ITS failed above $n = 200$.

Kanban and **FMS:** the superiority of SDD over MDD is clearly affirmed when variables have a large domain. In these models, the number of variables is fixed, but the variable domains grow in $O(n)$. In both these models, the complexity in Smart is near exponential where we obtain a low order polynomial.

For **Kanban**, in ITS/SDD we use the natural encoding that synchronizes four instances of a single type, yielding 4 variables of domain $O(n^3)$. However, an alternate flatter partitioning is used in Smart benchmarks, with 16 variables of domain size $O(n)$. This encoding limits the potential number of arcs per node, as the "rough" partition fails past $n \approx 50$.

Papers by Ciardo et al. on Smart (e.g. [5]) that compare the performances of a "rough" partition versus a "fine" one conclude that a fine partition is better for MDD. As our experiments section 4 show, this is not the case for SDD. Smart is based on MDD, which allow to represent integer domain variables rather than on SDD. Thus, when the size of the set of local states of a component grows (i.e. the domain of variables is large), performance is degraded.

This is due to the creation of MDD nodes having a large set of arcs. In contrast SDD are resistant to large local state spaces, as arcs are fused when they lead to the same successor node. Thus the number of arcs per node is not directly related to the number of local states, but rather to the complexity of state dependencies between components.

6   Conclusion

This paper investigates how hierarchical Set Decision Diagrams (SDD) may provide an efficient representation of the state space of composite systems. We have introduced *Instantiable Transition Systems* (ITS) as a framework to define formalisms in a way that allows to take advantage of the characteristics of SDD.

Our contributions are : *1)* we have defined encoding strategies which allow take advantage of the regularity of systems, *2)* ITS allow to capture this regularity using the notions of *type* and *instance*, *3)* the definition of ITS is generic allowing to adapt it to many formalisms.

Experimentation shows that our solution is competitive with existing symbolic solutions such as SMART or NuSMV. In the case of very regular models, we can even obtain a exponential compression factor in both time and memory.

Hierarchical Set Decision Diagrams are available as an open source C++ library `http://ddd.lip6.fr`, which has already been used to build several efficient model-checkers.

Definition of heuristics allowing to detect the regularity of a model and automatically propose an appropriate encoding strategy is left to future work.

References

1. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Formal Methods in System Design 10(2/3), 171–206 (1997)
2. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 341–353. Springer, Heidelberg (1999)
3. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
4. Burch, J.R., Clarke, E.M., McMillan, K.L.: Symbolic model checking: 10^{20} states and beyond. Information and Computation (Special issue for best papers from LICS90) 98(2), 153–181 (1992)
5. Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state-space generation. Formal Methods in System Design 31(1), 63–100 (2007)

6. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003)
7. Ciardo, G., Miner, A.S.: Implicit data structures for logic and stochastic systems analysis. SIGMETRICS Perform. Eval. Rev. 32(4), 4–9 (2005)
8. Couvreur, J.-M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.-A.: Data Decision Diagrams for Petri Net Analysis. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 1–101. Springer, Heidelberg (2002)
9. Couvreur, J.-M., Thierry-Mieg, Y.: Hierarchical Decision Diagrams to Exploit Model Structure. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 443–457. Springer, Heidelberg (2005)
10. Donatelli, S., Franceschinis, G.: The PSR Methodology: Integrating Hardware and Software Models. In: Proceedings of the 17th International Conference on Application and Theory of Petri Nets, London, UK, pp. 133–152. Springer, London (1996)
11. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical Set Decision Diagrams and Automatic Saturation. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 211–230. Springer, Heidelberg (2008)
12. Wang, F.: Formal verification of timed systems: A survey and perspective. IEEE 92(8) (August 2004)

Büchi Complementation and Size-Change Termination[*]

Seth Fogarty and Moshe Y. Vardi[**]

Department of Computer Science, Rice University, Houston, TX
{sfogarty,vardi}@cs.rice.edu

Abstract. We compare tools for complementing nondeterministic Büchi automata with a recent termination-analysis algorithm. Complementation of Büchi automata is a key step in program verification. Early constructions using a Ramsey-based argument have been supplanted by rank-based constructions with exponentially better bounds. In 2001 Lee et al. presented the size-change termination (SCT) problem, along with both a reduction to Büchi automata and a Ramsey-based algorithm. This algorithm strongly resembles the initial complementation constructions for Büchi automata.

We prove that the SCT algorithm is a specialized realization of the Ramsey-based complementation construction. Surprisingly, empirical analysis suggests Ramsey-based approaches are superior over the domain of SCT problems. Upon further analysis we discover an interesting property of the problem space that both explains this result and provides a chance to improve rank-based tools. With these improvements, we show that theoretical gains in efficiency are mirrored in empirical performance.

1   Introduction

The automata-theoretic approach to formal program verification reduces questions about program adherence to a specification to questions about language containment. Representing liveness, fairness, or termination properties requires finite automata that operate on infinite words. One automaton, $\mathcal{A}$, encodes the behavior of the program, while another automaton, $\mathcal{B}$, encodes the formal specification. To ensure adherence, verify that the intersection of $\mathcal{A}$ with the complement of $\mathcal{B}$ is empty. Thus a vital problem is constructing the complementary automata $\overline{\mathcal{B}}$. Finite automata on infinite words are classified by their acceptance condition and transition structure. We consider here nondeterministic Büchi automata, in which a run is accepting when it visits at least one accepting state infinitely often.

The first complementation constructions for nondeterministic Büchi automata employed a Ramsey-based combinatorial argument to partition infinite words into a finite set of regular languages. Proposed by Büchi in 1962 [3], this construction was shown in 1987 by Sistla, Vardi, and Wolper to be implementable with a blow-up of $2^{O(n^2)}$ [14]. This brought the complementation problem into singly-exponential blow-up, but left a gap with the $2^{\Omega(n \log n)}$ lower bound proved by Michel [11].

The gap was tightened in 1988, when Safra described a $2^{O(n \log n)}$ construction [13]. Work since then has focused on improving the practicality of $2^{O(n \log n)}$ constructions, either by providing simpler constructions, further tightening the bound, or improving the derived algorithms. In 2001, Kupferman and Vardi employed a rank-based analysis of Büchi automata to simplify complementation [9]. Recently Doyen and Raskin tightly integrated the rank-based construction with a subsumption relation to provide a complementation solver that scales to automata several orders of magnitude larger than previous tools [5].

Separately, in the context of of program termination analysis, Lee, Jones, and Ben-Amram presented the size-change termination (SCT) principle in 2001 [10]. This principle states that, for domains with well-founded values, if every infinite computation contains an infinitely decreasing value sequence, then no infinite computation is possible. Lee et al. describe a method of size-change termination analysis and reduce this problem to the containment of two Büchi automata. Stating the lack of efficient Büchi containment solvers, they also propose a Ramsey-based combinatorial solution that captures all possible call sequences in a finite set of graphs. The Lee, Jones, and Ben-Amram (LJB) algorithm was provided as a practical alternative to reducing the verification problem to Büchi containment, but bears a striking resemblance to the 1987 Ramsey-based complementation construction [14].

In this paper we show that the LJB algorithm for deciding SCT [10] is a specialized implementation of the 1987 Ramsey-based complementation construction [14]. We then empirically explore Lee et al.'s intuition that Ramsey-based algorithms are more practical than Büchi complementation tools on SCT problems. Initial experimentation does suggest that Ramsey-based tools are superior to rank-based tools on SCT problems. This is surprising, as the worst-case complexity of the LJB algorithm is significantly worse than that of rank-based tools. Investigating this discovery, we note that it is natural for SCT problems to be reverse-deterministic, and that for reverse-deterministic problems the worst-case bound for Ramsey-based algorithms matches that of the rank-based approach. This suggests improving the rank-based approach in the face of reverse determinism. We demonstrate that, indeed, reverse-deterministic automata have a maximum rank of 2, dramatically lowering the complexity of complementation to $2^{O(n)}$. Revisiting our experiments, we discover that with this improvement rank-based tools are superior on the domain of SCT problems.

2   Preliminaries

In this section we review the relevant details of the Büchi complementation and size-change termination, introducing along the way the notation used throughout this paper. An *nondeterministic Büchi automaton on infinite words* is a tuple $\mathcal{B} = \langle \Sigma, Q, Q^{in}, \rho, F \rangle$, where Σ is a finite nonempty alphabet, Q a finite nonempty set of states, $Q^{in} \subseteq Q$ a set of initial states, $F \subseteq Q$ a set of accepting states, and $\rho : Q \times \Sigma \rightarrow 2^Q$ a nondeterministic transition relation. We lift the ρ function to sets of states and words of arbitrary length in the usual fashion.

A *run* of a Büchi automaton $\mathcal{B}$ on a word $w \in \Sigma^\omega$ is a infinite sequence of states $q_0 q_1 ... \in Q^\omega$ such that $q_0 \in Q^{in}$ and, for every $i \geq 0$, we have $q_{i+1} \in \rho(q_i, w_i)$. A run is *accepting* iff $q_i \in F$ for infinitely many $i \in \mathbb{N}$. A word $w \in \Sigma^\omega$ is accepted by $\mathcal{B}$ if

there is an accepting run of $\mathcal{B}$ on w. The words accepted by $\mathcal{B}$ form the language of $\mathcal{B}$, denoted by $L(\mathcal{B})$. A *path* in $\mathcal{B}$ from q to r is a finite subsection of a run beginning in q and ending in r. A path is *accepting* if some state in the path is in F.

A Büchi automaton $\mathcal{A}$ is contained in a Büchi automaton $\mathcal{B}$ iff $L(\mathcal{A}) \subseteq L(\mathcal{B})$, which can be checked by verifying that the intersection of $\mathcal{A}$ with the complement $\overline{\mathcal{B}}$ of $\mathcal{B}$ is empty: $L(\mathcal{A}) \cap L(\overline{\mathcal{B}}) = \emptyset$. We know that the language of an automaton is non-empty iff there are states $q \in Q^{in}$, $r \in F$ such that there is a path from q to r and an accepting path from r to itself. The initial path is called the prefix, and the combination of the prefix and cycle is called a *lasso* [16]. Further, the intersection of two automata can be constructed, having a number of states proportional to the product of the number states of the original automata [4]. Thus, the most computationally demanding step is constructing the complement of $\mathcal{B}$. In the formal verification field, existing work has focused on the simplest form of containment testing, universality testing, where $\mathcal{A}$ is the universal automaton [5,15].

2.1 Ramsey-Based Universality

When Büchi introduced these automata in 1962, he described a complementation construction involving a Ramsey-based combinatorial argument. We describe an improved implementation presented in 1987. To construct the complement of $\mathcal{B}$, where $Q = \{q_0, ..., q_{n-1}\}$, we construct a set $\widetilde{Q}_\mathcal{B}$ whose elements capture the essential behavior of $\mathcal{B}$. Each element corresponds to an answer to the following question. Given a finite nonempty word w, for every two states $q, r \in Q$: is there a path in $\mathcal{B}$ from q to r over w, and is some such path accepting?

Define $Q' = Q \times \{0, 1\} \times Q$, and $\widetilde{Q}_\mathcal{B}$ to be the subset of $2^{Q'}$ whose elements do not contain both $\langle q, 0, r \rangle$ and $\langle q, 1, r \rangle$ for any q and r. Each element of $\widetilde{Q}_\mathcal{B}$ is a $\{0, 1\}$-arc-labeled graph on Q. An arc represents a path in $\mathcal{B}$, and the label is 1 if the path is accepting. Note that there are 3^{n^2} such graphs. With each graph $\widetilde{g} \in \widetilde{Q}_\mathcal{B}$ we associate a language $L(\widetilde{g})$, the set of words for which the answer to the posed question is the graph encoded by $\widetilde{g}$.

Definition 1. *Let $\widetilde{g} \in \widetilde{Q}_\mathcal{B}$ and $w \in \Sigma^+$. Then $w \in L(\widetilde{g})$ iff, for all pairs of states $q, r \in Q$:*

(1) $\langle q, a, r \rangle \in \widetilde{g}$, $a \in \{0, 1\}$, iff there is a path in $\mathcal{B}$ from q to r over w.
(2) $\langle q, 1, r \rangle \in \widetilde{g}$ iff there is an accepting path in $\mathcal{B}$ from q to r over w.

The languages $L(\widetilde{g})$, for the graphs $\widetilde{g} \in \widetilde{Q}_\mathcal{B}$, form a partition of Σ^+. With this partition of Σ^+ we can devise a finite family of ω-languages that cover Σ^ω. For every $\widetilde{g}, \widetilde{h} \in \widetilde{Q}_\mathcal{B}$, let Y_{gh} be the ω-language $L(\widetilde{g}) \cdot L(\widetilde{h})^\omega$. We say that a language Y_{gh} is *proper* if Y_{gh} is non-empty, $L(\widetilde{g}) \cdot L(\widetilde{h}) \subseteq L(\widetilde{g})$, and $L(\widetilde{h}) \cdot L(\widetilde{h}) \subseteq L(\widetilde{h})$. There are a finite, if exponential, number of such languages. A Ramsey-based argument shows that every infinite string belongs to a language of this form, and that $\overline{L(\mathcal{B})}$ can be expressed as the union of languages of this form.

Lemma 1. [3,14]
(1) $\Sigma^\omega = \bigcup \{Y_{gh} \mid Y_{gh} \text{ is proper}\}$
(2) For $\widetilde{g}, \widetilde{h} \in \widetilde{Q}_\mathcal{B}$, either $Y_{gh} \cap L(\mathcal{B}) = \emptyset$ or $Y_{gh} \subseteq L(\mathcal{B})$.

(3) $\overline{L(\mathcal{B})} = \bigcup \{Y_{gh} \mid Y_{gh}$ *is proper and* $Y_{gh} \cap L(\mathcal{B}) = \emptyset\}$.

To obtain the complementary Büchi automaton $\overline{\mathcal{B}}$, Sistla et al. construct, for each $\widetilde{g} \in \widetilde{Q}_{\mathcal{B}}$, a deterministic automata on finite words, $\mathcal{B}_g$, that accepts exactly $L(\widetilde{g})$. Using the automata $\mathcal{B}_g$, one can then construct the complementary automaton $\overline{\mathcal{B}}$ [14]. We can then use a lasso-finding algorithm on $\overline{\mathcal{B}}$ to prove the emptiness of $\overline{\mathcal{B}}$, and thus the universality of $\mathcal{B}$. We can avoid an explicit lasso search, however, by employing the rich structure of the graphs in $\widetilde{Q}_{\mathcal{B}}$. For every two graphs $\widetilde{g}, \widetilde{h} \in \widetilde{Q}_{\mathcal{B}}$, determine if Y_{gh} is proper. If Y_{gh} is proper, test if it is contained in $L(\mathcal{B})$ by looking for a lasso with a prefix in $\widetilde{g}$ and a cycle in $\widetilde{h}$. $\mathcal{B}$ is universal if every proper Y_{gh} is so contained.

Lemma 2. *Given an Büchi automaton $\mathcal{B}$ and the set of graphs $\widetilde{Q}_{\mathcal{B}}$,*

(1) $\mathcal{B}$ is universal iff, for every proper Y_{gh}, $Y_{gh} \subseteq L(\mathcal{B})$.
(2) Let $\widetilde{g}, \widetilde{h} \in \widetilde{Q}_{\mathcal{B}}$ be two graphs where Y_{gh} is proper. $Y_{gh} \subseteq L(\mathcal{B})$ iff there exists $q \in Q^{in}$, $r \in Q$, $a \in \{0, 1\}$ where $\langle q, a, r \rangle \in \widetilde{g}$ and $\langle r, 1, r \rangle \in \widetilde{h}$.

Lemma 2 yields a PSPACE algorithm to determine universality [14]. Simply check each $\widetilde{g}, \widetilde{h} \in \widetilde{Q}_{\mathcal{B}}$. If Y_{gh} is both proper and not contained in $L(\mathcal{B})$, then the pair $(\widetilde{g}, \widetilde{h})$ provide a counterexample to the universality of $\mathcal{B}$. If no such pair exists, the automaton must be universal.

2.2 Rank-Based Complementation

If a Büchi automaton $\mathcal{B}$ does not accept a word w, then every run of $\mathcal{B}$ on w must eventually cease visiting accepting states. The rank-based construction uses a notion of ranks to track the progress of each possible run towards fair termination. A *level ranking* for an automaton $\mathcal{B}$ with n states is a function $f : Q \to \{0...2n, \bot\}$, such that if $q \in F$ then $f(q)$ is even or $\bot$. Let a be a letter in Σ and f, f' be two level rankings f. Say that f covers f' under a when for all q and every $q' \in \rho(q, a)$, if $f(q) \neq \bot$ then $f'(q') \leq f(q)$; i.e. no transition between f and f' on a increases in rank. Let F_r be the set of all level rankings.

If $\mathcal{B} = \langle \Sigma, Q, Q^{in}, \rho, F \rangle$ is a Büchi automaton, define $KV(\mathcal{B})$ to be the automaton $\langle \Sigma, F_r \times 2^Q, \langle f_{in}, \emptyset \rangle, \rho', F_r \times \{\emptyset\} \rangle$, where

- $f_{in}(q) = 2n$ for each $q \in Q^{in}$, $\bot$ otherwise.
- Define $\rho' : \langle F_r \times 2^Q \rangle \times \sigma \to 2^{\langle F_r \times 2^Q \rangle}$ to be
  - If $o \neq \emptyset$ then $\rho'(\langle f, o \rangle, \sigma) =$
    $\{\langle f', o' \setminus d \rangle \mid f$ covers f' under σ, $o' = \rho(o, \sigma)$, $d = \{q \mid f'(q)$ odd$\}\}$.
  - If $o = \emptyset$ then $\rho'(\langle f, o \rangle, \sigma) =$
    $\{\langle f', f' \setminus d \rangle \mid f$ covers f' under a, $d = \{q \mid f'(q)$ odd$\}\}$.

Lemma 3. [9] *For every Büchi automaton $\mathcal{B}$, $L(KV(\mathcal{B})) = \overline{L(\mathcal{B})}$.*

An algorithm seeking to refute the universality of $\mathcal{B}$ can look for a lasso in the state-space of $KV(\mathcal{B})$. The strongest algorithm performing this search takes advantage of the presence of a subsumption relation in the KV construction: one state $\langle f, o \rangle$ subsumes another $\langle f', o' \rangle$ iff $f'(x) \leq f(x)$ for every $x \in Q$, $o' \subseteq o$, and $o = \emptyset$ iff

$o' = \emptyset$. When computing the backward-traversal lasso-finding fixed point, it is sufficient to represent a set of states with the maximal elements under this relation. Further, the predecessor operation over a single state and letter results in at most two incomparable elements. This algorithm has scaled to automata an order of magnitude larger than other approaches [5].

2.3   Size-Change Termination

In [10] Lee et al. proposed the size-change termination (SCT) principle for programs: "If every infinite computation would give rise to an infinitely decreasing value sequence, then no infinite computation is possible." The original presentation concerned a first-order pure functional language, where every infinite computation arises from an infinite call sequence and values are always passed through a sequence of parameters.

Proving that a program is size-change terminating is done in two phases. The first extracts from a program a set of size-change graphs, $\mathcal{G}$, containing guarantees about the relative size of values at each function call site. The second phase, and the phase we focus on, analyzes these graphs to determine if every infinite call sequence has a value that descends infinitely along a well-ordered set. For a discussion of the abstraction of language semantics, refer to [10].

Definition 2. *A* size-change graph *(SCG) from function f_1 to function f_2, written $G : f_1 \to f_2$, is a bipartite $\{0,1\}$-arc-labeled graph from the parameters of f_1 to the parameters of f_2, where $G \subseteq P(f_1) \times \{0,1\} \times P(f_2)$ does not contain both $x \xrightarrow{1} y$ and $x \xrightarrow{0} y$.*

Size-change graphs capture information about a function call. An arc $x \xrightarrow{1} y$ indicates that the value of x in the function f_1 is strictly greater than the value passed as y to function f_2. An arc $x \xrightarrow{0} y$ indicates that x's value is greater than or equal to the value given to y. We assume that all call sites in a program are reachable from the entry points of the program[1].

A *size-change termination* (SCT) problem is a tuple $L = \langle H, P, C, \mathcal{G} \rangle$, where H is a set of functions, P a mapping from each function to its parameters, C a set of call sites between these functions, and $\mathcal{G}$ a set of SCGs for C. A call site is written $c : f_1 \to f_2$ for a call to function f_2 occurring in the body of f_1. The size-change graph for a call site $c : f_1 \to f_2$ is written as G_c. Given a SCT problem L, a *call sequence* in L is a infinite sequence $cs = c_0, c_1, \ldots \in C^{\omega}$, such that there exists a sequence of functions $f_0, f_1, \ldots$ where $c_0 : f_0 \to f_1, c_1 : f_1 \to f_2 \ldots$. A *thread* in a call sequence $c_0, c_1, \ldots$ is a connected sequence of arcs, $x \xrightarrow{a} y, y \xrightarrow{b} z, \ldots$, beginning in some call c_i such that $x \xrightarrow{a} y \in G_{c_i}, y \xrightarrow{b} z \in G_{c_{i+1}}, \ldots$. We say that L is *size-change terminating* if every call sequence contains a thread with infinitely many 1-labeled arcs. Note that a thread need not begin at the start of a call sequence. A sequence must terminate if *any* well-founded value decreases infinitely often. Therefore threads can begin at any function call, in any parameter. We call this the *late-start property* of SCT problems, and revisit it in Section 3.2.

---

[1] The implementation provided by Lee et al. [10] also make this assumption, and in the presence of unreachable functions size-change termination may be undetectable.

Every call sequence can be represented as a word in C^ω, and a SCT problem reduced to the containment of two ω-languages. The first language $Flow(L) = \{cs \in C^\omega \mid cs \text{ is a call sequence}\}$, contains all call sequences. The second language, $Desc(L) = \{cs \in Flow(L) \mid \text{some thread in } cs \text{ has infinitely many 1-labeled arcs}\}$, contains only call sequences that guarantee termination. A SCT problem L is size-change terminating if and only if $Flow(L) \subseteq Desc(L)$.

Lee et al. [10] describe two Büchi automata, $\mathcal{A}_{Flow(L)}$ and $\mathcal{A}_{Desc(L)}$, that accept these languages. $\mathcal{A}_{Flow(L)}$ is simply the call graph of the program. $\mathcal{A}_{Desc(L)}$ waits in a copy of the call graph and nondeterministically chooses the beginning point of a descending thread. From there it ensures that a 1-labeled arc is taken infinitely often. To do so, it keeps two copies of each parameter, and transitions to the accepting copy only on a 1-labeled arc. Lee et al. prove that $L(\mathcal{A}_{Flow(L)}) = Flow(L)$, and $L(\mathcal{A}_{Desc(L)}) = Desc(L)$.

Definition 3. [2]
$\mathcal{A}_{Flow(L)} = \langle C, H, H, \rho_F, H \rangle$, where
  – $\rho_F(f_1, c) = \{f_2 \mid c : f_1 \rightarrow f_2\}$

$\mathcal{A}_{Desc(L)} = \langle C, Q_1 \cup H, H, \rho_D, F \rangle$, where
  – $Q_1 = \{\langle x, r \rangle \mid f \in H, x \in P(f), r \in \{1, 0\}\}$,
  – $\rho_D(f_1, c) = \{f_2 \mid c : f_1 \rightarrow f_2\} \cup \{\langle x, r \rangle \mid c : f_1 \rightarrow f_2, x \in P(f_2), r \in \{0, 1\}\}$
  – $\rho_D(\langle x, r \rangle, c) = \{\langle x', r' \rangle \mid x \xrightarrow{r'} x' \in \mathcal{G}_c\}$,
  – $F = \{\langle x, 1 \rangle \mid f \in H, x \in P(f)\}$

Using the complementation constructions of either Section 2.1 or 2.2 and a lasso-finding algorithm, we can determine the containment of $\mathcal{A}_{Flow(L)}$ in $\mathcal{A}_{Desc(L)}$. Lee et al. propose an alternative graph-theoretic algorithm, employing SCGs to encode descent information about entire call sequences. A notion of composition is used, where a call sequence $c_0...c_{n-1}$ has a thread from x to y if and only if the composition of the SCGs for each call, $G_{c_0}; ...; G_{c_{n-1}}$, contains the arc $x \xrightarrow{a} y$. The closure S of $\mathcal{G}$ under the composition operation is then searched for a counterexample describing an infinite call sequence with no infinitely descending thread.

Definition 4. *Let* $G : f_1 \rightarrow f_2$ *and* $G' : f_2 \rightarrow f_3$ *be two SCGs. Their composition* $G; G'$ *is defined as* $G'' : f_1 \rightarrow f_3$ *where:*

$$G'' = \{x \xrightarrow{1} z \mid x \xrightarrow{a} y \in G, y \xrightarrow{b} z \in G', y \in P(f_2), a = 1 \text{ or } b = 1\}$$
$$\cup \{x \xrightarrow{0} z \mid x \xrightarrow{0} y \in G, y \xrightarrow{0} z \in G', y \in P(f_2), \text{ and}$$
$$\forall y', r, r' . x \xrightarrow{r} y' \in G \wedge y' \xrightarrow{r'} z \in G' \text{ implies } r = r' = 0\}$$

Theorem 1. [10] *A SCT problem* $L = \langle H, P, C, \mathcal{G} \rangle$ *is* not *size-change terminating iff* S, *the closure of* $\mathcal{G}$ *under composition, contains a SCG graph* $G : f \rightarrow f$ *such that* $G = G; G$ *and* G *does* not *contain an arc of the form* $x \xrightarrow{1} x$.

---

[2] The original LJB construction [10] restricted edges from functions to parameters to the 0-labeled parameters. This was changed to simplify Section 3.3. The modification does not change the accepted language.

Theorem 1, whose proof uses a Ramsey-based argument, yields an algorithm that determines the size-change termination of an SCT problem $L = \langle H, P, C, \mathcal{G} \rangle$ by ensuring the absence of a counterexample in the closure of $\mathcal{G}$ under composition. First, use an iterative algorithm to build the closure set S: initialize S as $\mathcal{G}$; and for every $G : f_1 \rightarrow f_2$ and $G' : f_2 \rightarrow f_3$ in S, include the composition $G; G'$ in S. Second, check every $G : f_1 \rightarrow f_1 \in S$ to ensure that if G is idempotent, i.e. $G = G; G$, then G contains an arc of the form $x \xrightarrow{1} x$.

3   Size-Change Termination and Ramsey-Based Containment

The Ramsey-based test of Section 2.1 and the LJB algorithm of Section 2.3 bear a more than passing similarity. In this section we bridge the gap between the Ramsey-based universality test and the LJB algorithm, by demonstrating that the LJB algorithm is a specialized realization of the Ramsey-based containment test. This first requires developing a Ramsey-based framework for Büchi -containment testing.

3.1   Ramsey-Based Containment with Supergraphs

To test the containment of a Büchi automaton $\mathcal{A}$ in a Büchi automaton $\mathcal{B}$, we could construct the complement of $\mathcal{B}$ using either the Ramsey-based or rank-based construction, compute the intersection automaton of $\mathcal{A}$ and $\overline{\mathcal{B}}$, and search this intersection automaton for a lasso. With universality, however, we avoided directly constructing $\overline{\mathcal{B}}$ by exploiting the structure of states in the Ramsey-based construction (see Lemma 2). We demonstrate a similar test for containment.

Consider two automata, $\mathcal{A} = \langle \Sigma, Q_\mathcal{A}, Q_\mathcal{A}^{in}, \rho_\mathcal{A}, F_\mathcal{A} \rangle$ and $\mathcal{B} = \langle \Sigma, Q_\mathcal{B}, Q_\mathcal{B}^{in}, \rho_\mathcal{B}, F_\mathcal{B} \rangle$. When testing the universality of $\mathcal{B}$, any word not in $L(\mathcal{B})$ is a sufficient counterexample. To test $L(\mathcal{A}) \subseteq L(\mathcal{B})$ we must restrict our search to the subset of Σ^ω accepted by $\mathcal{A}$. In Section 2.1, we defined a set $\widetilde{Q}_\mathcal{B}$, which provides a family of languages that covers Σ^ω (see Lemma 1). We now define a set, $\widehat{Q}_{\mathcal{A},\mathcal{B}}$, which provides a family of languages covering $L(\mathcal{A})$.

We first define $\bar{Q}_\mathcal{A} = Q_\mathcal{A} \times Q_\mathcal{A}$ to capture the connectivity in $Q_\mathcal{A}$. An element $\bar{g} = \langle q, r \rangle \in \bar{Q}_\mathcal{A}$ is a single arc asserting the existence of a path in $\mathcal{A}$ from q to r. With each arc we associate a language, $L(\bar{g})$. Given a word $w \in \Sigma^+$, say that $w \in L(\langle q, r \rangle)$ iff there is a path in $\mathcal{A}$ from q to r over w. Define $\widehat{Q}_{\mathcal{A},\mathcal{B}}$ as $\bar{Q}_\mathcal{A} \times \widetilde{Q}_\mathcal{B}$. The elements of $\widehat{Q}_{\mathcal{A},\mathcal{B}}$, called *supergraphs*, are pairs consisting of an arc from $\bar{Q}_\mathcal{A}$ and a graph from $\widetilde{Q}_\mathcal{B}$. Each element simultaneously captures all paths in $\mathcal{B}$ and a single path in $\mathcal{A}$. The language $L(\langle \bar{g}, \widetilde{g} \rangle)$ is then $L(\bar{g}) \cap L(\widetilde{g})$. For convenience, we implicitly take $\widehat{g} = \langle \bar{g}, \widetilde{g} \rangle$, and say $\langle q, a, r \rangle \in \widehat{g}$ when $\langle q, a, r \rangle \in \widetilde{g}$.

The languages $L(\widehat{g})$, $\widehat{g} \in \widehat{Q}_{\mathcal{A},\mathcal{B}}$, cover all finite subwords of $L(\mathcal{A})$. With them we define a finite family of ω-languages that cover $L(\mathcal{A})$. Given $\widehat{g}, \widehat{h} \in \widehat{Q}_{\mathcal{A},\mathcal{B}}$, let Z_{gh} be the ω-language $L(\widehat{g}) \cdot L(\widehat{h})^\omega$. Z_{gh} is called *proper* if: (1) Z_{gh} is non-empty; (2) $\bar{g} = \langle q, r \rangle$ and $\bar{h} = \langle r, r \rangle$ where $q \in Q_\mathcal{A}^{in}$ and $r \in F_\mathcal{A}$; (3) $L(\widehat{g}) \cdot L(\widehat{h}) \subseteq L(\widehat{g})$ and $L(\widehat{h}) \cdot L(\widehat{h}) \subseteq L(\widehat{h})$. We note that Z_{gh} is non-empty if $L(\widehat{g})$ and $L(\widehat{h})$ are non-empty, and that, by the second condition, every proper Z_{gh} is contained in $L(\mathcal{A})$.

Lemma 4. *Let $\mathcal{A}$ and $\mathcal{B}$ be two Büchi automata, and $\widehat{Q}_{\mathcal{A},\mathcal{B}}$ be the corresponding set of supergraphs.*

(1) $L(\mathcal{A}) = \bigcup\{Z_{gh} \mid Z_{gh}$ is proper$\}$
(2) For all proper Z_{gh}, either $Z_{gh} \cap L(\mathcal{B}) = \emptyset$ or $Z_{gh} \subseteq L(\mathcal{B})$
(3) $L(\mathcal{A}) \subseteq L(\mathcal{B})$ iff every proper language $Z_{gh} \subseteq L(\mathcal{B})$.
(4) Let $\widehat{g}, \widehat{h}$ be two supergraphs such that Z_{gh} is proper. $Z_{gh} \subseteq L(\mathcal{B})$ iff there exists $q \in Q_{\mathcal{B}}^{in}$, $r \in Q_{\mathcal{B}}$, $a \in \{0,1\}$ such that $\langle q, a, r \rangle \in \widehat{g}$ and $\langle r, 1, r \rangle \in \widehat{h}$.

In an analogous fashion to Section 2.1, we can use supergraphs to test the containment of two automata, $\mathcal{A}$ and $\mathcal{B}$. Search all pairs of supergraphs, $\widehat{g}, \widehat{h} \in \widehat{Q}_{\mathcal{A},\mathcal{B}}$ for a pair that is both proper and for which there does not exist a $q \in Q_{\mathcal{B}}^{in}$, $r \in Q_{\mathcal{B}}$, $a \in \{0,1\}$ such that $\langle q, a, r \rangle \in \widehat{g}$ and $\langle r, 1, r \rangle \in \widehat{h}$. Such a pair is a counterexample to containment. If no such pair exists, then $L(\mathcal{A}) \subseteq L(\mathcal{B})$. We call this search the *double-graph search*, to distinguish from later algorithms for which a counterexample is a single graph.

The double-graph search faces difficulty on two fronts. First, the number of potential supergraphs is very large. Secondly, checking language nonemptiness is an exponentially difficult problem. To address these problems we construct only supergraphs with non-empty languages. Borrowing the notion of composition from Section 2.3 allows us to use exponential space to compute exactly the needed supergraphs. We start with graphs corresponding to single letters and compose them until we reach closure. The resulting subset of $\widehat{Q}_{\mathcal{A},\mathcal{B}}$, written $\widehat{Q}_{\mathcal{A},\mathcal{B}}^f$, contains exactly the supergraphs with non-empty languages. In addition to removing the need to check for emptiness, composition allows us to test the sole remaining aspect of properness, language containment, in time polynomial in the size of the supergraphs.

3.2 Strongly Suffix Closed Languages

Theorem 1 suggests that, for some languages, a cycle implies the existence of a lasso. For Büchi automata of such languages, it is sufficient, when disproving containment, to search for a graph $\widehat{h} \in \widehat{Q}_{\mathcal{B}}$, where $\widehat{h}; \widehat{h} = \widehat{h}$, with no arc $\langle r, 1, r \rangle$. This single-graph search reduces the complexity of our algorithm significantly. What enables this in size-change termination is the late-start property: threads can begin at any point. We here define the class of automata amenable to this optimization, beginning with universality for simplicity.

In size-change termination, an accepting cycle can start at any point. Thus the arc $\langle r, 1, r \rangle \in \widetilde{h}$ does not need an explicit matching prefix $\langle q, a, r \rangle$ in some $\widetilde{g}$. In the context of universality, we can apply this method when it is safe to add or remove arbitrary prefixes of a word. To describe these languages we extend the standard notion of *suffix closure*. A language L is suffix closed when, for every $w \in L$, every suffix of w is in L.

Definition 5. *A language L is* strongly suffix closed *if it is suffix closed and for every $w \in L$, $w_1 \in \Sigma^+$, we have that $w_1 w \in L$.*

Lemma 5. *Let $\mathcal{B}$ be an Büchi automaton where every state in Q is reachable and $L(\mathcal{B})$ is strongly suffix closed. $\mathcal{B}$ is not universal iff the set of supergraphs with non-empty languages, $\widetilde{Q}_{\mathcal{B}}^f$, contains a graph $\widetilde{h} = \widetilde{h}; \widetilde{h}$ with no arc of the form $\langle r, 1, r \rangle$.*

To extend this notion to handle containment questions $L_1 \subseteq L_2$, we restrict our focus to words in L_1. Instead of requiring L_2 to be closed under arbitrary prefixes, L_2 need only be closed under prefixes that keep the word in L_1.

Definition 6. *A language L_2 is* strongly suffix closed *with respect to L_1 when L_2 is suffix closed and, for every $w \in L_1 \cap L_2$, $w_1 \in \Sigma^+$, if $w_1w \in L_1$ then $w_1w \in L_2$.*

Lemma 6. *Let $\mathcal{A}$ and $\mathcal{B}$ be two Büchi automata where $Q_\mathcal{A}^{in} = Q_\mathcal{A}$,[3] every state in $Q_\mathcal{B}$ is reachable, and $L(\mathcal{B})$ is strongly suffix closed with respect to $L(\mathcal{A})$. Then $L(\mathcal{A}) \not\subseteq L(\mathcal{B})$ iff $\widehat{Q}_{\mathcal{A},\mathcal{B}}^f$ contains a supergraph $\widehat{h} = \langle\langle s, s\rangle, \widetilde{h}\rangle$ where $s \in F_\mathcal{A}$, $\widehat{h}; \widehat{h} = \widehat{h}$ and there is no arc $\langle r, 1, r\rangle \in \widehat{h}$.*

Lemma 6 provides a simplified test for the containment of $\mathcal{A}$ in $\mathcal{B}$ when $L(\mathcal{B})$ is strongly suffix closed with respect to $L(\mathcal{A})$. Search all supergraphs in $\widehat{Q}_{\mathcal{A},\mathcal{B}}$ for an supergraph $\widehat{h}$ where $\widehat{h}; \widehat{h} = \widehat{h}$ that does not contain an arc of the form $\langle r, 1, r\rangle$. The presence of this counterexample refutes containment, and the absence of such a supergraph proves containment. We call this search the *single-graph search*.

3.3   From Ramsey-Based Containment to Size-Change Termination

We can now delve into the connection between the LJB algorithm for size-change termination and the Ramsey-based containment test. SCGs of the LJB algorithm are direct analogues of supergraphs in the Ramsey-based containment test of $\mathcal{A}_{Flow(L)}$ and $\mathcal{A}_{Desc(L)}$.

Noting that the LJB algorithm examines single SCGs G where $G = G; G$, we show that for an SCT problem $L = \langle H, P, C, \mathcal{G}\rangle$ the conditions of Lemma 6 are met. First, every state in $\mathcal{A}_{Flow(L)}$ is an initial state. Second, every function in L is reachable, and so every state in $\mathcal{A}_{Desc(L)}$ is reachable.[4] Finally, the late-start property is precisely $Desc(L)$ being strongly suffix closed with respect to $Flow(L)$. Therefore we can use the single-graph search.

Consider supergraphs in $\widehat{Q}_{\mathcal{A}_{Flow(L)},\mathcal{A}_{Desc(L)}}$. The state space of $\mathcal{A}_{Flow(L)}$ is the set of functions H, and the state space of $\mathcal{A}_{Desc(L)}$ is the union of H and Q_1, the set of all $\{0, 1\}$-labeled parameters. A supergraph in $\widehat{Q}_{\mathcal{A}_{Flow(L)},\mathcal{A}_{Desc(L)}}$ thus comprises an arc $\langle q, r\rangle$ in H and a $\{0, 1\}$-labeled graph $\widetilde{g}$ over $H \cup Q_1$. The arc asserts the existence of a call path from q to r, and the graph $\widetilde{g}$ captures the relevant information about corresponding paths in $\mathcal{A}_{Desc(L)}$.

These supergraphs are almost the same as SCGs, $G : q \to r$. Aside from notational differences, both contain an arc, which asserts the existence of a call path between two functions, and a $\{0, 1\}$-labeled graph. There are vertices in both graphs that correspond to parameters of functions, and arcs between two such vertices describe a thread between the corresponding parameters. The analogy falls short, however, on three points:

(1) In SCGs, vertices are always parameters of functions. In supergraphs, vertices can be either parameters of functions or function names.

---

[3] With a small amount of work, the restriction that $Q_\mathcal{A}^{in} = Q_\mathcal{A}$ can be relaxed to the requirement that $L(\mathcal{A})$ be suffix closed.

[4] In the original reduction, 1-labeled parameters may not have been reachable.

(2) In SCGs, vertices are unlabeled. In supergraphs, vertices are labeled either 0 or 1.

(3) In SCGs, only vertices corresponding to parameters of two specific functions are present. In supergraphs, vertices corresponding to every parameters of every functions exist.

We show, in turn, that each difference is an opportunity to specialize the Ramsey-based containment algorithm.

(1) No functions in H are accepting for $\mathcal{A}_{Desc(L)}$, and once we transition out of H into Q_1 we can never return to H. Therefore vertices corresponding to function names can never be part of a descending arc $\langle r, 1, r \rangle$. Since we only search $\widehat{J}$ for a cycle $\langle r, 1, r \rangle$, we can simplify supergraphs in $\widehat{Q}_{\mathcal{A}_{Flow(L)}, \mathcal{A}_{Desc(L)}}$ by removing all vertices corresponding to functions.

(2) The labels on parameters are the result of encoding a Büchi edge acceptance condition in a Büchi state acceptance condition automaton, and can be dropped from supergraphs with no loss of information. Consider an arc $\langle \langle f, a \rangle, b, \langle g, c \rangle \rangle$. If b is 1, we know the corresponding thread contains a descending arc. The value of c tells us if the final arc in the thread is descending, but which arc is descending is irrelevant. Thus it is safe to simplify supergraphs in $\widehat{Q}_{\mathcal{A}_{Flow(L)}, \mathcal{A}_{Desc(L)}}$ by removing labels on parameters.

(3) While all parameters are states in $\mathcal{A}_{Desc(L)}$, each supergraph describes threads in a call sequence between two functions. There are no threads in this call sequence between parameters of other functions, and so no supergraph with a non-empty language has arcs between the parameters of other functions. We can thus simplify supergraphs in $\widehat{Q}_{\mathcal{A}_{Flow(L)}, \mathcal{A}_{Desc(L)}}$ by removing all vertices corresponding to parameters of other functions.

We can specialize the Ramsey-based containment algorithm for $L(\mathcal{A}_{Flow(L)}) \subseteq L(\mathcal{A}_{Desc(L)})$ in two ways. First, by Lemma 6 we know that $Flow(L) \subseteq Desc(L)$ if and only if $\widehat{Q}_{\mathcal{A}_{Flow(L)}, \mathcal{A}_{Desc(L)}}$ contains an idempotent graph $\widehat{g} = \widehat{g}; \widehat{g}$ with no arc of the form $\langle r, 1, r \rangle$. Secondly, we can simplify supergraphs in $\widehat{Q}_{\mathcal{A}_{Flow(L)}, \mathcal{A}_{Desc(L)}}$ by removing the labels on parameters and keeping only the vertices associated with appropriate parameters. The simplifications of supergraphs whose languages contain single characters are in one-to-one corresponding with $\mathcal{G}$, the initial set of SCGs. As every state in $Flow(L)$ is accepting, every idempotent supergraph can serve as a counterexample. Therefore $Desc(L) \subseteq Flow(L)$ if and only if the closure of the set of simplified supergraphs under composition contains an idempotent supergraph with no arc of the form $\langle r, 1, r \rangle$. This is precisely the algorithm provided by Theorem 1.

4   Empirical Analysis

All the Ramsey-based algorithms presented in Section 2.3 have worst-case running times that are exponentially slower than those of the rank-based algorithms. We now compare existing, Ramsey-based, SCT tools tools to a rank-based Büchi containment solver on the domain of SCT problems.

4.1   Towards an Empirical Comparison

To facilitate a fair comparison, we briefly describe two improvements to the algorithms presented above. First, in constructing the analogy between SCGs in the LJB algorithm

and supergraphs in the Ramsey-based containment algorithm, we noticed that super-graphs contain vertices for every parameter, while SCGs contain only vertices corresponding to parameters of relevant functions. These vertices are states in $\mathcal{A}_{Desc(L)}$. While we can specialize the Ramsey-based test to avoid them, Büchi containment solvers might suffer. These states duplicate information. As we already know which functions each supergraph corresponds to, there is no need for each vertex to be unique to a specific function.

The extra states emerge because $Desc(L)$ only accepts strings that are contained in $Flow(L)$. But the behavior of $\mathcal{A}_{Desc(L)}$ on strings not in $Flow(L)$ is irrelevant to the question of $Flow(L) \subseteq Desc(L)$, and we can replace the names of parameters in $\mathcal{A}_{Desc(L)}$ with their location in the argument list. By using this observation, we can simplify the reduction from SCT problems to Büchi containment problems. Experimental results demonstrate that these changes do improve performance.

Second, in [2], Ben-Amram and Lee present a polynomial approximation of the LJB algorithm for SCT. To facilitate a fair comparison, they optimize the LJB algorithm for SCT by using subsumption to remove certain SCGs when computing the closure under composition. This suggests that the single-graph search of Lemma 6 can also employ subsumption. When computing the closure of a set of supergraphs under compositions, we can ignore elements when they are conservatively approximated, or subsumed, by other elements. Intuitively, a supergraph $\widehat{g}$ conservatively approximates another super-graph $\widehat{h}$ when it is strictly harder to find a 1-labeled sequence of arcs through $\widehat{g}$ than through $\widehat{h}$. When the right arc can be found in $\widehat{g}$, then it also occurs in $\widehat{h}$. If $\widehat{g}$ does not have a satisfying arc, then we already have a counterexample supergraph. Formally, given two graphs $\widehat{g}, \widehat{h} \in \widehat{Q}_{\mathcal{A},\mathcal{B}}$ where $\bar{g} = \bar{h}$, say that $\widehat{g}$ *conservatively approximates* $\widehat{h}$, written $\widehat{g} \preceq \widehat{h}$, when for every arc $\langle q, a, r \rangle \in \widehat{g}$ there is an arc $\langle q, a', r \rangle \in \widehat{h}$, where if $a = 1$ then $a' = 1$. Note that conservative approximation is a transitive relation. In order to safely employ conservative approximation as a subsumption relation, we replace the search for a single arc in idempotent graphs with a search for a strongly connected component in all graphs. Extending this relationship to the double-graph search is an open problem.

4.2   Experimental Results

All experiments were performed on a Dell Optiplex GX620 with a single 1.7Ghz Intel Pentium 4 CPU and 512 MB. Each tool was given 3500 seconds, a little under one hour, to complete each task.

Tools: The formal-verification community has implemented rank-based tools in order to measure the scalability of various approaches. The programming-languages community has implemented several Ramsey-based SCT tools. We use the best-of-breed rank-based tool, **Mh**, developed by Doyen and Raskin [5], that leverages a subsumption relation on ranks. We expanded the Mh tool to handle Büchi containment problems with arbitrary languages, thus implementing the full containment-checking algorithm presented in their paper.

We use two Ramsey-based tools. **SCTP** is a direct implementation of the LJB algorithm of Theorem 1, written in Haskell [7]. We have extended SCTP to reduce SCT problems to Büchi containment problems, using either Definition 3 or our improved

reduction. **sct/scp** is an optimized C implementation of the SCT algorithm, which uses the subsumption relation of Section 4.1 [2].

Problem Space: Existing experiments on the practicality of SCT solvers focus on examples extracted from the literature [2]. We combine examples from a variety of sources [1,2,7,8,10,12,17]. The time spent reducing SCT problems to Büchi automata never took longer than 0.1 seconds and was dominated by I/O. Thus this time was not counted. We compared the performance of the rank-based Mh solver on the derived Büchi containment problems to the performance of the existing SCT tools on the original SCT problems. If an SCT problem was solved in all incarnations and by all tools in less than 1 second, the problem was discarded as uninteresting. Unfortunately, of the 242 SCT problems derived from the literature, only 5 prove to be interesting.

Experiment Results: Table 1 compares the performance of the rank-based Mh solver against the performance of the existing SCT tools, displaying which problems each tool could solve, and the time taken to solve them. Of the interesting problems, both SCTP and Mh could only complete 3. On the other hand, sct/scp completed all of them, and had difficulty with only one problem.

Table 1. SCT problem completion time by tool

Problem	SCTP (s)	Mh (s)	sct/scp (s)
ex04 [2]	1.58	Time Out	1.39
ex05 [2]	Time Out	Time Out	227.7
ms [7]	Time Out	0.1	0.02
gexgcd [7]	0.55	14.98	0.023
graphcolour2 [8]	0.017	3.18	0.014

The small problem space makes it difficult to draw firm conclusions, but it is clear that Ramsey-based tools are comparable to rank-based tools on SCT problems: the only tool able to solve all problems was Ramsey based. This is surprising given the significant difference in worst-case complexity, and motivates further exploration.

5   Reverse-Determinism

In the previous section, the theoretical gap in performance between Ramsey and rank-based solutions was not reflected in empirical analysis. Upon further investigation, it is revealed that a property of the domain of SCT problems is responsible. Almost all problems, and every difficult problem, in this experiment have SCGs whose vertices have an in-degree of at most 1. This property was first observed by Ben-Amram and Lee in their analysis of SCT complexity [2]. After showing why this property explains the performance of Ramsey-based algorithms, we explore why this property emerges and argue that it is a reasonable property for SCT problems to possess. Finally, we improve the rank-based algorithm for problems with this property.

As stated above, all interesting SCGs in this experiment have vertices with at most one incoming edge. In analogy to the corresponding property for automaton, we call this property of SCGs *reverse-determinism*. Given a set of reverse-deterministic SCGs $\mathcal{G}$, we observe three consequences. First, a reverse-deterministic SCG can have no more

than n arcs: one entering each vertex. Second, there are only $2^{O(n \log n)}$ possible such combinations of n arcs. Third, the composition of two reverse-deterministic SCGs is also reverse-deterministic. Therefore every element in the closure of $\mathcal{G}$ under composition is also reverse-deterministic. These observations imply that the closure of $\mathcal{G}$ under composition contains at most $2^{O(n \log n)}$ SCGs. This reduces the worst-case complexity of the LJB algorithm to $2^{O(n \log n)}$. In the presence of this property, the massive gap between Ramsey-based algorithms and rank-based algorithms vanishes, helping to explain the surprising strength of the LJB algorithm.

Lemma 7. *When operating on reverse-deterministic SCT problems, the LJB algorithm has a worst-case complexity of $2^{O(n \log n)}$.*

It is not a coincidence that all SCT problems considered possess this property. As noted in [2], straightforward analysis of functional programs generates only reverse-deterministic problems. In fact, every tool we examined is only capable of producing reverse-deterministic SCT problems. To illuminate the reason for this, imagine a SCG $G : f \to g$ where f has two parameters, x and y, and g the single parameter z. If G is not reverse deterministic, this implies both x and y have arcs, labeled with either 0 or 1, to z. This would mean that z's value is both *always* smaller than or equal to x and *always* smaller than or equal to y. In order for this to occur, we would need a min operation that returns the smaller of two elements. For the case of lists, for example, min would return the shorter of two lists. This is not a common operation, and none of the size-change analyzers were designed to discover such properties of functions.

We now consider the rank-based approach to see if it can benefit from reverse-determinism. We say that an automaton is *reverse-deterministic* when no state has two incoming arcs labeled with the same character. Formally, an automaton is reverse-deterministic when, for each state q and character a, there is at most one state p such that $q \in \rho(p, a)$. Given a reverse-deterministic SCT problem L, both $\mathcal{A}_{Flow(L)}$ and $\mathcal{A}_{Desc(L)}$ are reverse-deterministic. As a corollary to the above, the Ramsey-based complementation construction has a worst-case complexity of $2^{O(n \log n)}$ for reverse deterministic automata. Examining the rank-based approach, we note that with reverse-deterministic automata we do not have to worry about multiple paths to a state. Thus a maximum rank of 2, rather than $2n$, suffices to prove termination of every path, and the worst-case bound of the rank-based construction improves to $2^{O(n)}$.

Lemma 8. *Given a reverse-deterministic Büchi automaton $\mathcal{B}$ with n states, there exists an automaton $\mathcal{B}'$ with $2^{O(n)}$ states such that $L(\mathcal{B}') = \overline{L(\mathcal{B})}$.*

In light of this discovery, we revisit the experiments and again compare rank and Ramsey-based approaches on SCT problems. This time we tell Mh, the rank-based solver, that the problems have a maximum rank of 2. Table 2 compares the running time of Mh and sct/scp on the five most difficult problems. As before, time taken to reduce SCT problems to automata containment problems was not counted.

While our problem space is small, the theoretical worst-case bounds of Ramsey and rank-based approach appears to be reflected in the table. The Ramsey-based sct/scp completes some problems more quickly, but in the worst cases, ex04 and ex05, performs significantly more slowly than Mh. It is worth noting, however, that the benefits of

Table 2. SCT problem completion time times by tool, exploiting reverse-determinism

Problem	Mh (s)	sct/scp (s)
ex04	0.01	1.39
ex05	0.13	227.7
ms	0.1	0.02
gexgcd	0.39	0.023
graphcolour2	0.044	0.014

reverse-determinism on Ramsey-based approaches emerges automatically, while rank-based approaches must explicitly test for this property in order to exploit it.

6   Conclusion

In this paper we demonstrate that the Ramsey-based size-change termination algorithm proposed by Lee, Jones, and Ben-Amram [10] is a specialized realization of the 1987 Ramsey-based complementation construction [3,14]. With this link established, we compare rank-based and Ramsey-based tools on the domain of SCT problems. Initial experimentation revealed a surprising competitiveness of the Ramsey-based tools, and led us to further investigation. By exploiting reverse-determinism, we were able to demonstrate the superiority of the rank-based approach.

Our experiments operated on a very sparse space of problem, and still yielded two interesting observations. First, subsumption appears to be critical to the performance of Büchi complementation tools using both rank and Ramsey-based algorithms. It has already been established that rank-based tools benefit strongly from the use of subsumption [5]. Our results demonstrate that Ramsey-based tools also benefit from subsumption, and in fact experiments with removing subsumption from sct/scp seem to limit its scalability. Second, by exploiting reverse-determinism, we can dramatically improve the performance of both rank and Ramsey-based approaches to containment checking.

Our test space was unfortunately small, with only five interesting problems emerging. In [5,15], a space of random automata universality problems is used to provide a diverse problem domain. We plan to similarly generate a space of random SCT problems to provide a more informative problem space. Sampling this problem space is complicated by the low transition density of reverse-deterministic problems: in [5,15] the most interesting problems had a transition density of 2. Intrigued by the competitive performance of Ramsey-based solutions, we also intend to compare Ramsey and rank-based approaches on the domain of random universality problems.

On the theoretical side, we are interested in extending the subsumption relation present in sct/scp. It is not immediately clear how to use subsumption for problems that are not strongly suffix-closed. While arbitrary problems can be phrased as a single-graph search, doing so imposes additional complexity. Extending the subsumption relation to the double-graph search of Lemma 4 would simplify this solution greatly.

The effects of reverse-determinism on the complementation of automata bear further study. Reverse-determinism is not an obscure property, it is known that automata derived from LTL formula are reverse-deterministic [6]. As noted above, both rank and Ramsey-based approaches improves exponentially when operating on reverse-deterministic

automata. Further, Ben-Amram and Lee have defined SCP, a polynomial-time approximation algorithm for SCT. For a wide subset of SCT problems with restricted in degrees, including the set used in this paper, SCP is exact. In terms of automata, this property is similar, although perhaps not identical, to reverse-determinism. The presence of an exact polynomial algorithm for the SCT case suggests a interesting subset of Büchi containment problems may be solvable in polynomial time. The first step in this direction would be to determine what properties a containment problem must have to be solved in this fashion.

References

1. Daedalus, `http://www.di.ens.fr/~cousot/projects/DAEDALUS/`
2. Ben-Amram, A.M., Lee, C.: Program termination analysis in polynomial time. ACM Trans. Program. Lang. Syst 29(1) (2007)
3. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: ICLMPS, pp. 1–12. Stanford University Press (1962)
4. Choueka, Y.: Theories of automata on ω-tapes: A simplified approach. Journal of Computer and Systems Science 8, 117–141 (1974)
5. Doyen, L., Raskin, J.-F.: Improved algorithms for the automata-based approach to model-checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 451–465. Springer, Heidelberg (2007)
6. Emerson, A.E., Sistla, A.P.: Deciding full branching time logics. Information and Control 61(3), 175–201 (1984)
7. Frederiksen, C.C.: A simple implementation of the size-change termination principle. Tech. Rep. D-442, DIKU (2001)
8. Glenstrup, A.J.: Terminator ii: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen (June 1999)
9. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. Transactions on Computational Logic, 409–429 (2001)
10. Lee, C., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92 (2001)
11. Michel, M.: Complementation is more difficult with automata on infinite words. In: CNET, Paris (1988)
12. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 281–297. Springer, Heidelberg (2005)
13. Safra, S.: On the Complexity of ω-Automat. In: FOCS, pp. 319–327 (1988)
14. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 217–237. Springer, Heidelberg (1985)
15. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS, vol. 3835, pp. 396–411. Springer, Heidelberg (2005)
16. Vardi, M.Y.: Automata-theoretic model checking revisited. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 137–150. Springer, Heidelberg (2007)
17. Wahlstedt, D.: Detecting termination using size-change in parameter values. Master's thesis, Göteborgs Universitet (2000)

Learning Minimal Separating DFA's for Compositional Verification[⋆]

Yu-Fang Chen[1], Azadeh Farzan[2], Edmund M. Clarke[3], Yih-Kuen Tsay[1],
and Bow-Yaw Wang[4]

[1] National Taiwan University
[2] University of Toronto
[3] Carnegie Mellon University
[4] Academia Sinica

Abstract. Algorithms for learning a minimal separating DFA of two disjoint regular languages have been proposed and adapted for different applications. One of the most important applications is learning minimal contextual assumptions in automated compositional verification. We propose in this paper an efficient learning algorithm, called L^{Sep}, that learns and generates a minimal separating DFA. Our algorithm has a quadratic query complexity in the product of sizes of the minimal DFA's for the two input languages. In contrast, the most recent algorithm of Gupta *et al.* has an exponential query complexity in the sizes of the two DFA's. Moreover, experimental results show that our learning algorithm significantly outperforms all existing algorithms on randomly-generated example problems. We describe how our algorithm can be adapted for automated compositional verification. The adapted version is evaluated on the LTSA benchmarks and compared with other automated compositional verification approaches. The result shows that our algorithm surpasses others in 30 of 49 benchmark problems.

1 Introduction

Compositional verification is seen by many as a promising approach for scaling up Model Checking [8] to larger designs. In the approach, one applies a compositional inference rule to break the task of verifying a system down to the subtasks of verifying its components. The compositional inference rule is usually in the so-called *assume-guarantee* style. One widely used assume-guarantee rule, formulated from a language-theoretic view, is the following:

$$\frac{\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P) \quad \mathcal{L}(M_2) \subseteq \mathcal{L}(\mathcal{A})}{\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \subseteq \mathcal{L}(P)}$$

We assume that the behaviors of a system or component are characterized by a language and any desired property is also described as a language. The parallel composition of two components is represented by the intersection of the languages of the two components. A system (or component) satisfies a property if the language of the system (or component) is a subset of the language of the property. The above assume-guarantee rule then says that, to verify that the system composed of components M_1 and M_2 satisfies property P, one may instead verify the following two conditions: (1) component M_1 satisfies (guarantees) P under some contextual assumption $\mathcal{A}$ and (2) component M_2 satisfies the contextual assumption $\mathcal{A}$.

The main difficulty in applying assume-guarantee rules to compositional verification is the need of human intervention to find contextual assumptions. For the case where components and properties are given as regular languages, several automatic approaches have been proposed to find contextual assumptions [4,10] based on the machine learning algorithm L^* [2,17]. Following this line of research, there have been results for symbolic implementations [1,18], various optimization techniques [12,6], an extension to liveness properties [11], performance evaluation [9], and applications to problems such as component substitutability analysis [5]. However, all of the above suffer from the same problem: they do not guarantee finding a small assumption even if one exists. Though minimality of the assumption does not ensure better performance, we will show in this paper that it helps most of the time.

The problem of finding a minimal assumption for compositional verification can be reduced to the problem of finding a minimal separating DFA (deterministic finite automaton) of two disjoint regular languages [14]. A DFA $\mathcal{A}$ *separates* two disjoint languages L_1 and L_2 if its language $\mathcal{L}(\mathcal{A})$ contains L_1 and is disjoint from L_2 ($L_1 \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}) \cap L_2 = \emptyset$). The DFA $\mathcal{A}$ is *minimal* if it has the least number of states among all separating DFA's. Several approaches [14,16,13] have been proposed to find a minimal separating DFA automatically. However, all of those approaches are computationally expensive. In particular, the most recent algorithm of Gupta *et al.* [14] has an exponential query complexity in the sizes of the minimal DFA's of the two input languages.

In this paper we propose a more efficient learning algorithm, called L^{Sep}, that finds the aforementioned minimal separating DFA. The query complexity of our algorithm is quadratic in the product of the sizes of the two minimal DFA's for the two input languages. Moreover, our algorithm utilizes membership queries to accelerate learning and has a more compact representation of the samples collected from the queries. Experiments show that L^{Sep} significantly outperforms other algorithms on a large set of randomly-generated example problems.

We then give an adaptation of the L^{Sep} algorithm for automated compositional verification and evaluate its performance on the LTSA benchmarks [9]. The result shows that the adapted version of L^{Sep} surpasses other compositional verification algorithms on 30 of 49 benchmark problems. Besides automated compositional verification, algorithms for learning a minimal separating DFA have found other applications. For example, Grinchtein *et al.* [13] used such an al-

gorithm as the basis for *learning network invariants of parameterized systems*. Although we only discuss the application of L^{Sep} to automated compositional verification in this paper, the algorithm can certainly be adapted for other applications as well.

2   Preliminaries

An *alphabet* Σ is a finite set. A finite *string* over Σ is a finite sequence of elements from Σ. The empty string is represented by λ. The set of all finite strings over Σ is denoted by Σ^*, and Σ^+ is the set of all nonempty finite strings over Σ (so, $\Sigma^+ = \Sigma^* \backslash \{\lambda\}$). The length of string u is denoted by $|u|$ and $|\lambda| = 0$. For two strings $u = u_1 \ldots u_n$ and $v = v_1 \ldots v_m$ where $u_i, v_j \in \Sigma$, define the concatenation of the two strings as $uv = u_1 \ldots u_n v_1 \ldots v_m$. For a string u, u^n is recursively defined as uu^{n-1} with $u^0 = \lambda$. String concatenation is naturally extended to sets of strings where $S_1 S_2 = \{s_1 s_2 | \ s_1 \in S_1, s_2 \in S_2\}$. A string u is a *prefix* (respectively *suffix*) of another string v if and only if there exists a string $w \in \Sigma^*$ such that $v = uw$ (respectively $v = wu$). A set of strings S is called *prefix-closed* (respectively *suffix-closed*) if and only if for all $v \in S$, if u is a prefix (respectively suffix) of v, then $u \in S$.

A *deterministic finite automaton (DFA)* $\mathcal{A}$ is a tuple $(\Sigma, S, s_0, \delta, F)$, where Σ is an alphabet, S is a finite set of states, s_0 is the initial state, $\delta : S \times \Sigma \to S$ is the transition function, and $F \subseteq S$ is a set of *accepting* states. The transition function δ is extended to strings of any length in the natural way. A string u is accepted by $\mathcal{A}$ if and only if $\delta(s_0, u) \in F$. Define $\mathcal{L}(\mathcal{A}) = \{u \mid u$ is accepted by $\mathcal{A}\}$. A language $L \subseteq \Sigma^*$ is *regular* if and only if there exists a finite automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$. The notation $\overline{L}$ denotes the complement with respect to Σ^* of the regular language L. Let $|L|$ denote the number of states of the minimal DFA that recognizes L and $|\mathcal{A}|$ denote the number of states in the DFA $\mathcal{A}$.

Definition 1. *(Three-Valued Deterministic Finite Automata) A* 3-valued deterministic finite automaton (3DFA) $\mathcal{C}$ *is a tuple* $(\Sigma, S, s_0, \delta, Acc, Rej, Dont)$, *where* Σ, S, s_0, *and* δ *are as defined in a DFA.* S *is partitioned into three disjoint sets* Acc, Rej, *and* $Dont$. Acc *is the set of accepting states,* Rej *is the set of rejecting states, and* $Dont$ *is the set of don't care states.*

For a 3DFA $\mathcal{C} = (\Sigma, S, s_0, \delta, Acc, Rej, Dont)$, a string u is *accepted* if $\delta(s_0, u) \in Acc$, is *rejected* if $\delta(s_0, u) \in Rej$, and is a *don't care* string if $\delta(s_0, u) \in Dont$. Let $\mathcal{C}^+$ denote the DFA $(\Sigma, S, s_0, \delta, Acc \cup Dont)$, where all don't care states become accepting states, and $\mathcal{C}^-$ denote the DFA $(\Sigma, S, s_0, \delta, Acc)$, where all don't care states become rejecting states. By definition, we have that $\mathcal{L}(\mathcal{C}^-)$ is the set of accepted strings in $\mathcal{C}$ and $\overline{\mathcal{L}(\mathcal{C}^+)}$ is the set of rejected strings in $\mathcal{C}$.

A DFA $\mathcal{A}$ is *consistent with* a 3DFA $\mathcal{C}$ if and only if $\mathcal{A}$ accepts all strings that $\mathcal{C}$ accepts, and rejects all strings that $\mathcal{C}$ rejects. It follows that $\mathcal{A}$ accepts strings in $\mathcal{L}(\mathcal{C}^-)$ and rejects those in $\overline{\mathcal{L}(\mathcal{C}^+)}$, or equivalently, $\mathcal{L}(\mathcal{C}^-) \subseteq \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C}^+)$. A *minimal consistent* DFA of $\mathcal{C}$ is a DFA $\mathcal{A}$ which is consistent with $\mathcal{C}$ and has the

(a) A DFA $\mathcal{A}$ consistent with a 3DFA $\mathcal{C}$ (b) A DFA $\mathcal{A}$ separating L_1 and L_2
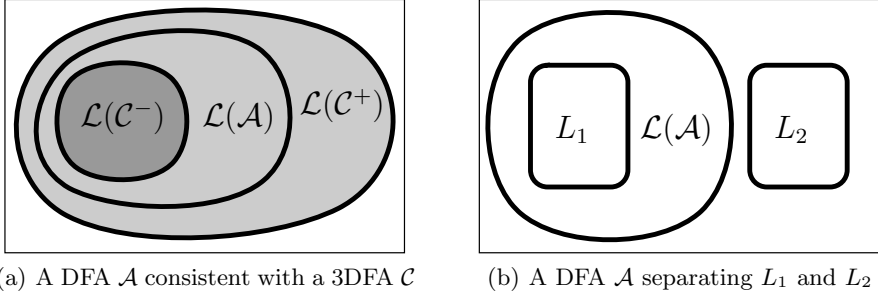
Fig. 1. Consistent and Separating DFA's

least number of states among all DFA's consistent with $\mathcal{C}$. Figure 1(a) illustrates a DFA $\mathcal{A}$ consistent with a 3DFA $\mathcal{C}$. In the figure, the bounding box is the set of all finite strings Σ^*. The dark shaded area represents $\mathcal{L}(\mathcal{C}^-)$. The union of the dark shaded area and the light shaded area represents $\mathcal{L}(\mathcal{C}^+)$. The DFA $\mathcal{A}$ is consistent with $\mathcal{C}$ as it accepts all strings in $\mathcal{L}(\mathcal{C}^-)$ and rejects those not in $\mathcal{L}(\mathcal{C}^+)$.

Given two disjoint regular languages L_1 and L_2, a *separating DFA* $\mathcal{A}$ for L_1 and L_2 satisfies $L_1 \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}) \cap L_2 = \emptyset$. It follows that $\mathcal{A}$ accepts all strings in L_1 and rejects those in L_2, or equivalently, $L_1 \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{L_2}$. We say a DFA $\mathcal{A}$ *separates* L_1 and L_2 if and only if $\mathcal{A}$ is a separating DFA for L_1 and L_2. A separating DFA is *minimal* if it has the least number of states among all separating DFA's for L_1 and L_2. Figure 1(b) shows a separating DFA $\mathcal{A}$ for L_1 and L_2.

A 3DFA $\mathcal{C}$ is *sound with respect to L_1 and L_2* if any DFA consistent with $\mathcal{C}$ separates L_1 and L_2. When the context is clear, we abbreviate "sound with respect to L_1 and L_2" simply as "sound". Figure 2(a) illustrates the condition when $\mathcal{C}$ is sound with respect to L_1 and L_2. Both $L_1 \subseteq \mathcal{L}(\mathcal{C}^-)$ and $\mathcal{L}(\mathcal{C}^+) \subseteq \overline{L_2}$ are true in this figure. Any DFA consistent with $\mathcal{C}$ accepts strings in $\mathcal{L}(\mathcal{C}^-)$ (the dark area) and possibly some strings in the light shaded area. Hence it accepts all strings in L_1 but none in L_2, i.e., it separates L_1 and L_2. Therefore, $\mathcal{C}$ is sound. Figure 2(c) illustrates the case that $\mathcal{C}$ is unsound. We can show that either $L_1 \not\subseteq \mathcal{L}(\mathcal{C}^-)$ or $\mathcal{L}(\mathcal{C}^+) \not\subseteq \overline{L_2}$ implies $\mathcal{C}$ is unsound. Assuming that we have $L_1 \not\subseteq \mathcal{L}(\mathcal{C}^-)$. It follows that there exists some string $u \in L_1$ that satisfies $u \notin \mathcal{L}(\mathcal{C}^-)$. The DFA $\mathcal{A}$ that recognizes $\mathcal{L}(\mathcal{C}^-)$ (the dark area) is consistent with $\mathcal{C}$. However, $\mathcal{A}$ is not a separating DFA for L_1 and L_2 because it rejects u, a string in L_1. We can then conclude that $\mathcal{C}$ is unsound. The case that $\mathcal{L}(\mathcal{C}^+) \not\subseteq \overline{L_2}$ can be shown to be unsound by a similar argument.

A 3DFA $\mathcal{C}$ is *complete with respect to L_1 and L_2* if any separating DFA for L_1 and L_2 is consistent with $\mathcal{C}$. Again, when the context is clear, we abbreviate "complete with respect to L_1 and L_2" as "complete". Figure 2(b) shows the situation when $\mathcal{C}$ is complete for L_1 and L_2. Any separating DFA for L_1 and L_2 accepts all strings in L_1 but none in L_2. Hence it accepts strings in $\mathcal{L}(\mathcal{C}^-)$ (the dark area) and possibly those in the light shaded area, i.e., it is consistent with $\mathcal{C}$. Therefore, $\mathcal{C}$ is complete. Figure 2(d) illustrates the case that $\mathcal{C}$ is incomplete.

(a) Soundness

(b) Completeness

(c) Unsoundness
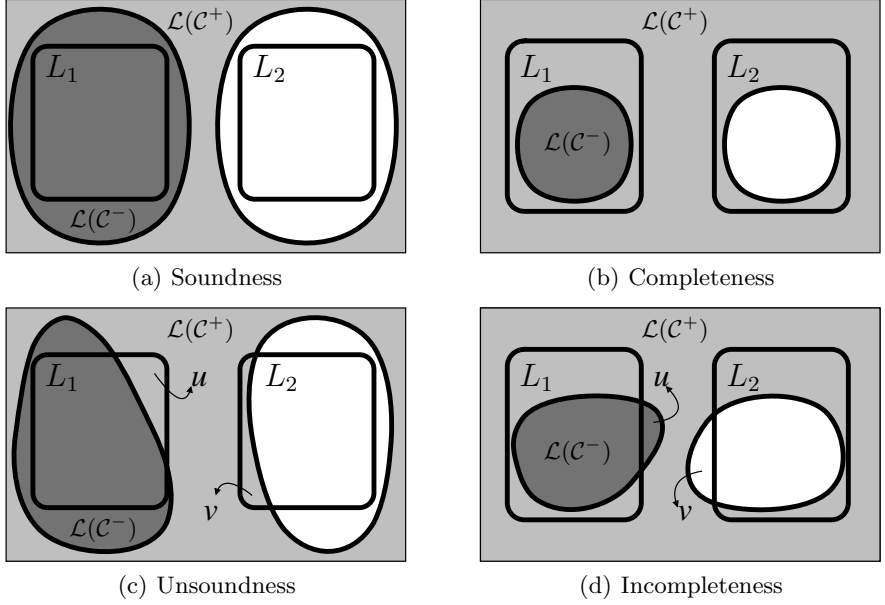
(d) Incompleteness

Fig. 2. Soundness and Completeness of a 3DFA $\mathcal{C}$

We can show that either $\mathcal{L}(\mathcal{C}^-) \not\subseteq L_1$ or $\overline{L_2} \not\subseteq \mathcal{L}(\mathcal{C}^+)$ implies $\mathcal{C}$ is incomplete. Assuming that we have $\mathcal{L}(\mathcal{C}^-) \not\subseteq L_1$. It follows that there exists some string $u \in \mathcal{L}(\mathcal{C}^-)$ that satisfies $u \notin L_1$. The DFA $\mathcal{A}$ that recognizes L_1 is a separating DFA for L_1 and L_2. However, $\mathcal{A}$ is not consistent with $\mathcal{C}$ because $\mathcal{A}$ rejects u, a string in $\mathcal{L}(\mathcal{C}^-)$. We can then conclude that $\mathcal{C}$ is incomplete. The case that $\overline{L_2} \not\subseteq \mathcal{L}(\mathcal{C}^+)$ can be shown to be incomplete by a similar argument.

Proposition 1. *Let L_1 and L_2 be regular languages and $\mathcal{C}$ be a 3DFA. Then*

1. *$\mathcal{C}$ is sound if and only if $L_1 \subseteq \mathcal{L}(\mathcal{C}^-)$ and $\mathcal{L}(\mathcal{C}^+) \subseteq \overline{L_2}$;*
2. *$\mathcal{C}$ is complete if and only if $\mathcal{L}(\mathcal{C}^-) \subseteq L_1$ and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}^+)$.*

3   Overview of Learning a Minimal Separating DFA

Given two disjoint regular languages L_1 and L_2, our task is to find a minimal DFA $\mathcal{A}$ that separates L_1 and L_2, namely $L_1 \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{L_2}$. Our key idea is to use a 3DFA as a succinct representation for the samples collected from L_1 and L_2. Exploiting the three possible acceptance outcomes of a 3DFA (accept, reject, and don't care), we encode strings from L_1 and L_2 in a 3DFA $\mathcal{C}$ as follows. All strings of L_1 are accepted by $\mathcal{C}$ and all strings in L_2 are rejected by $\mathcal{C}$. The remaining strings take $\mathcal{C}$ into don't care states. Observe that for any DFA $\mathcal{A}$, the following two conditions are equivalent: (1) $\mathcal{A}$ is consistent with $\mathcal{C}$, which means $\mathcal{A}$ accepts all accepted strings in $\mathcal{C}$ and rejects all rejected strings in $\mathcal{C}$. (2) $\mathcal{A}$ separates L_1 and L_2, which means $\mathcal{A}$ accepts all strings in L_1 and rejects all strings in L_2.

It follows that DFA's consistent with $\mathcal{C}$ and those separating L_1 and L_2 in fact coincide. We therefore reduce the problem of finding the minimal separating DFA for L_1 and L_2 to the problem of finding the minimal DFA consistent with the 3DFA $\mathcal{C}$.

By Proposition 1, $\mathcal{C}$ is both *sound* and *complete* with respect to L_1 and L_2 because $L_1 = \mathcal{L}(\mathcal{C}^-)$, the accepted strings in $\mathcal{C}$, and $L_2 = \overline{\mathcal{L}(\mathcal{C}^+)}$, the rejected strings in $\mathcal{C}$.
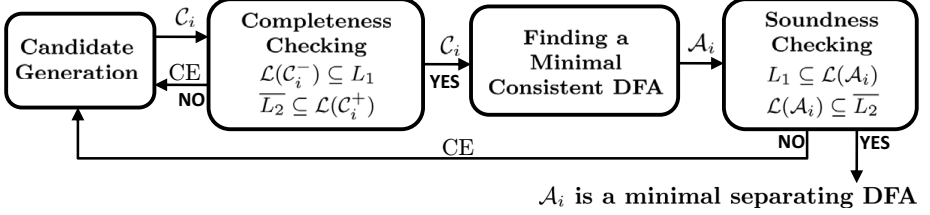
Fig. 3. Learning a Minimal Separating DFA – Overview

Figure 3 depicts the flow of our algorithm. The candidate generation step is performed by the *candidate generator*, which produces a series of candidate 3DFA's $\mathcal{C}_i$ targeting the 3DFA $\mathcal{C}$ using an extension of L^*. The *completeness checking* step examines whether $\mathcal{C}_i$ is complete with respect to L_1 and L_2. If $\mathcal{C}_i$ is incomplete, a counterexample is returned to the *candidate generator* to refine the next conjecture. Otherwise, $\mathcal{C}_i$ is complete, and the next step is to compute a minimal DFA $\mathcal{A}_i$ consistent with $\mathcal{C}_i$.

The following lemma characterizing the *sizes* of the minimal consistent DFA $\mathcal{A}_i$ and minimal separating DFA's for L_1 and L_2:

Lemma 1. *Let $\widehat{\mathcal{A}}$ be a minimal separating DFA of L_1 and L_2, and $\mathcal{A}_i$ be a minimal DFA consistent with $\mathcal{C}_i$. If $\mathcal{C}_i$ is complete, then $|\widehat{\mathcal{A}}| \geq |\mathcal{A}_i|$.*

Proof. By completeness, any separating DFA of L_1 and L_2 is consistent with $\mathcal{C}_i$. Hence the minimal separating DFA $\widehat{\mathcal{A}}$ is a DFA consistent with $\mathcal{C}_i$. Because $\mathcal{A}_i$ is the *minimal* DFA consistent with $\mathcal{C}_i$, we have $|\widehat{\mathcal{A}}| \geq |\mathcal{A}_i|$.   □

Finally, we check if $\mathcal{A}_i$ separates L_1 and L_2, i.e., $L_1 \subseteq \mathcal{L}(\mathcal{A}_i)$ and $\mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. If $\mathcal{A}_i$ is a separating DFA for L_1 and L_2, together with Lemma 1, we can conclude that $\mathcal{A}_i$ is a *minimal* separating DFA for L_1 and L_2. Note that even if $\mathcal{C}_i$ is unsound, it is still possible that a minimal consistent DFA of $\mathcal{C}_i$ separates L_1 and L_2. It follows that L^{Sep} may find a minimal separating DFA before the *candidate generator* produces the *sound* and *complete* 3DFA.

If $\mathcal{A}_i$ is not a separating DFA for L_1 and L_2, we get a counterexample to the *soundness* of $\mathcal{C}_i$ (will be described in the next section) and then send it to the *candidate generator* to refine the next conjecture. *Candidate generator* is guaranteed to converge to the *sound* and *complete* 3DFA, hence, our algorithm is guaranteed to find the minimal separating DFA and terminate.

4   The L^{Sep} Algorithm

L^{Sep} is an active[1] learning algorithm which computes a minimal separating DFA for two disjoint regular languages L_1 and L_2. It assumes a teacher that answers the following two types of queries:

- **membership queries** where the teacher returns *true* if the given string w is in L_1, *false* if w is in L_2, and *don't care* otherwise, and
- **containment queries** where the teacher solves language containment problems of the following four types: (i) $L_1 \subseteq \mathcal{L}(A_i)$, (ii) $\mathcal{L}(A_i) \subseteq L_1$, (iii) $\overline{L_2} \subseteq \mathcal{L}(A_i)$, and (iv) $\mathcal{L}(A_i) \subseteq \overline{L_2}$. The teacher returns "YES" if the containment holds, and "NO" with a counterexample otherwise, where A_i is a conjecture DFA.

As sketched in Section 3, the L^{Sep} algorithm performs the following steps to find a minimal separating DFA $\mathcal{A}$ for the languages L_1 and L_2 iteratively.

Candidate Generation

The candidate generation step is performed by the *candidate generator*, which extends the observation table in L^* [17] to allow entries with don't cares. An *observation table* $\langle S, E, T \rangle$ is a triple of a prefix-closed set S of strings, a set E of distinguishing strings, and a function T from $(S \cup S\Sigma) \times E$ to $\{+, -, ?\}$; see Figure 4 for an example. Let $\alpha \in S \cup S\Sigma$ and $\beta \in E$. The function T maps $\pi = (\alpha, \beta)$ to $+$ if $\alpha\beta \in L_1$; it maps π to $-$ if $\alpha\beta \in L_2$; otherwise T maps π to $?$. In the observation table of Figure 4, the entry for (ba, b) is $+$ because the string $bab \in L_1{}^2$.

	λ	b
λ	$-$	?
b	?	?
ba	$-$	$+$
bab	$+$	$+$
a	$-$	?
bb	?	?
baa	$-$	$+$
$baba$	$-$	?
$babb$	$+$	$+$

Fig. 4. An Observation Table and Its Corresponding 3DFA. The square node denotes a don't care state.

The *candidate generator* constructs the observation table by posing membership queries. It generates a 3DFA $\mathcal{C}_i$ based on the observation table. If the 3DFA $\mathcal{C}_i$ is unsound or incomplete, the *candidate generator* expands the observation table by extracting distinguishing strings from counterexamples and then generates another conjecture 3DFA. Let n be the size of the minimal sound and complete 3DFA and m be the length of the longest counterexample returned by containment queries. The *candidate generator* is guaranteed to find a sound and complete 3DFA with $O(n^2 + n \log m)$ membership queries. Moreover, it generates at most $n - 1$ incorrect 3DFA's. We refer the reader to [7] for details.

---

[1] A learning algorithm is *active* if it can actively query the teacher to label samples; otherwise, it is *passive*.

[2] Here $L_1 = (a^*b^+a^+b^+)(a^+b^+a^+b^+)^*$ and $L_2 = a^*(b^*a^+)^*$.

Completeness Checking

The L^{Sep} algorithm finds the minimal DFA separating L_1 and L_2 by computing the minimal DFA consistent with $\mathcal{C}_i$. To make sure all separating DFA's for L_1 and L_2 are considered, the L^{Sep} algorithm checks whether $\mathcal{C}_i$ is *complete*.

By Proposition 1, checking completeness reduces to checking whether $\mathcal{L}(\mathcal{C}_i^-) \subseteq L_1$ and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}_i^+)$, which can be done by containment queries. L^{Sep} first builds the DFA's $\mathcal{C}_i^+$ and $\mathcal{C}_i^-$. It then submits the containment queries $\mathcal{L}(\mathcal{C}_i^-) \subseteq L_1$ and $\overline{L_2} \subseteq \mathcal{L}(\mathcal{C}_i^+)$. If either of these queries fails, a counterexample is sent to the *candidate generator* to refine $\mathcal{C}_i$. Note that several iterations between *candidate generation* and *completeness checking* may be needed to find a complete 3DFA.

Finding a Minimal Consistent DFA

After the *completeness checking*, the next step is to compute a minimal DFA consistent with $\mathcal{C}_i$. We reduce the problem to the minimization problem of incompletely specified finite state machines [15]. The L^{Sep} algorithm translates the 3DFA $\mathcal{C}_i$ into an incompletely specified finite state machine $\mathcal{M}$. It then invokes the algorithm in [15] to obtain a minimal finite state machine $\mathcal{M}_i$ consistent with $\mathcal{M}$. Finally, $\mathcal{M}_i$ is converted to a DFA $\mathcal{A}_i$.

Soundness Checking

After the minimal DFA $\mathcal{A}_i$ consistent with $\mathcal{C}_i$ is computed, L^{Sep} verifies whether $\mathcal{A}_i$ separates L_1 and L_2 by the containment queries $L_1 \subseteq \mathcal{L}(\mathcal{A}_i)$ and $\mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. There are three possible outcomes:

- $L_1 \subseteq \mathcal{L}(\mathcal{A}_i) \subseteq \overline{L_2}$. Hence, $\mathcal{A}_i$ is in fact a separating DFA for L_1 and L_2. By Lemma 1, $\mathcal{A}_i$ is a *minimal* separating DFA for L_1 and L_2.
- $L_1 \nsubseteq \mathcal{L}(\mathcal{A}_i)$. There is a string $u \in L_1 \setminus \mathcal{L}(\mathcal{A}_i)$. Moreover, we have $\mathcal{L}(\mathcal{A}_i) \supseteq \mathcal{L}(\mathcal{C}_i^-)$ because $\mathcal{A}_i$ is consistent with $\mathcal{C}_i$. Therefore, $u \in L_1 \setminus \mathcal{L}(\mathcal{C}_i^-)$. By Proposition 1, u is a counterexample to the soundness of $\mathcal{C}_i$. It is sent to the *candidate generator* to refine the 3DFA in the next iteration.
- $\mathcal{L}(\mathcal{A}_i) \nsubseteq \overline{L_2}$. There is a string $v \in \mathcal{L}(\mathcal{A}_i) \setminus \overline{L_2}$. The string v is in fact a counterexample to the soundness of $\mathcal{C}_i$ by an analogous argument. It is sent to the *candidate generator* as well.

4.1   Correctness

The following theorem states the correctness of the L^{Sep} algorithm.

Theorem 1. *The L^{Sep} algorithm terminates and outputs a minimal separating DFA for L_1 and L_2.*

Proof. The statement follows from the following observations:

1. Each iteration of the L^{Sep} algorithm terminates.
2. If the minimal consistent DFA (submitted to soundness checking) separates L_1 and L_2, L^{Sep} terminates and returns a minimal separating DFA.

3. If the minimal consistent DFA does not separate L_1 and L_2, a counterexample to the soundness of C_i is sent to the *candidate generator*.
4. Because of 3, the *candidate generator* will eventually converge to the sound and complete 3DFA C defined in Section 3. In this case, the minimal consistent DFA is a minimal separating DFA for L_1 and L_2. Hence L^{Sep} terminates when C is found.   □

4.2 Complexity Analysis

We now estimate the number of queries used in the L^{Sep} algorithm. Lemma 2 states an upper bound on the size of the minimal sound and complete 3DFA (a proof can be found in [7]). By Lemma 2, the *query complexity* of L^{Sep} is established in Theorem 2.

Lemma 2. *Let B_i be the minimal DFA accepting the regular language L_i for $i = 1, 2$. The size of the minimal 3DFA C that accepts all strings in L_1 and rejects all strings in L_2 is smaller than $|B_1| \times |B_2|$.*

Theorem 2. *Let B_i be the minimal DFA accepting the regular language L_i for $i = 1, 2$. The L^{Sep} algorithm uses at most $O((|B_1| \times |B_2|)^2 + (|B_1| \times |B_2|) \log m)$ membership queries and $4(|B_1| \times |B_2|) - 1$ containment queries to learn a minimal separating DFA for L_1 and L_2, where m is the length of the longest counterexample returned by the teacher.*

Proof. Let C be a minimal 3DFA that accepts all strings in L_1 and rejects all strings in L_2. The *candidate generator* takes at most $O(|C|^2 + |C| \log m)$ membership queries and proposes at most $|C| - 1$ incorrect conjecture 3DFA's to L^{Sep}. By Lemma 2, the size of C is smaller than $|B_1| \times |B_2|$. It follows that the L^{Sep} algorithm takes $O((|B_1| \times |B_2|)^2 + (|B_1| \times |B_2|) \log m)$ membership queries and $4(|B_1| \times |B_2|) - 1$ containment queries (for each conjecture 3DFA, L^{Sep} uses at most 2 containment queries to check completeness and 2 containment queries to check soundness) to learn a minimal separating DFA in the worst case.   □

5 Automated Compositional Verification

We discuss how to adapt L^{Sep} to the context of automated compositional verification. The adapted version is referred to as "adapted L^{Sep}". We first explain how to reduce the problem of finding a minimal assumption in assume-guarantee reasoning to the problem of finding a minimal separating automaton. We then show how adapted L^{Sep} handles the case in which the system violates the property and introduce heuristics to improve the efficiency of the adapted algorithm.

Finding a minimal assumption in assume-guarantee reasoning: Suppose we want to use the following assume-guarantee rule to verify if the system composed of two components M_1 and M_2 satisfies a property P:

$$\frac{\mathcal{L}(M_2) \subseteq \mathcal{L}(\mathcal{A}) \quad \mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P)}{\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \subseteq \mathcal{L}(P)}$$

The second premise, $\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P)$, in the rule can be rewritten as $\mathcal{L}(\mathcal{A}) \subseteq \overline{(\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)})}^3$. Therefore, the two premises can be summarized as

$$\mathcal{L}(M_2) \subseteq \mathcal{L}(\mathcal{A}) \subseteq \overline{\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}}$$

This immediately translates the problem of finding a minimal assumption in assume-guarantee reasoning to the problem of finding a minimal separating automaton of the two languages $\mathcal{L}(M_2)$ and $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}$. Therefore, if the system composed of M_1 and M_2 satisfies the property P, L^{Sep} can be used to find a contextual assumption $\mathcal{A}$ that is needed by the assume-guarantee rule[4].

The case when the system violates the property: The adapted L^{Sep} algorithm handles the case that the system violates the property as follows:

1. A membership query on a string v returns *true*, *false*, or *don't care* in the same way as the original L^{Sep} algorithm.
2. In addition, it returns *fail* if v is in both input languages. If *fail* is returned by a query, the adapted L^{Sep} algorithm terminates and reports v as a witness that the two languages are not disjoint, i.e., the property is violated.[5]
3. When a conjecture query returns a counterexample w, the adapted L^{Sep} algorithm submits a membership query on w. If *fail* is not returned by the query, the algorithm proceeds as usual.

The following lemma states the correctness of the adapted L^{Sep} algorithm (a proof can be found in [7]):

Lemma 3. *If* $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P)$, *eventually the* fail *result will be returned by a membership query.*

Heuristics for efficiency: Minimizing a 3DFA is computationally expensive. In the context of automated compositional verification, we do not need to insist on finding a minimal solution. A heuristic algorithm that finds a small assumption with lower cost may be preferred. The adapted L^{Sep} algorithm uses the following heuristic to build a "reduced" DFA consistent with a 3DFA.

We first use Paull and Unger's algorithm [15] to find the sets of "maximal" compatible states[6], which are the candidates for the states in the reduced DFA. Consider an example shown in Figure 5. We have $Q_1 = \{s_0, s_1\}$, $Q_2 = \{s_0, s_2\}$, $Q_3 = \{s_0, s_3, s_4\}$.

---

[3] It can be done using the following steps: $\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P) \Leftrightarrow (\mathcal{L}(M_1) \cap \mathcal{L}(\mathcal{A})) \cap \overline{\mathcal{L}(P)} = \emptyset \Leftrightarrow \mathcal{L}(\mathcal{A}) \cap (\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}) = \emptyset \Leftrightarrow \mathcal{L}(\mathcal{A}) \subseteq \overline{(\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)})}$.

[4] The reduction was first observed by Gupta *et al.* [14].

[5] The facts that *the system violates the property* and *the two input languages are not disjoint* are equivalent to each other, which can be proved as follows: $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \nsubseteq \mathcal{L}(P) \Leftrightarrow \mathcal{L}(M_1) \cap \mathcal{L}(M_2) \cap \overline{\mathcal{L}(P)} \neq \emptyset \Leftrightarrow \mathcal{L}(M_2) \cap (\mathcal{L}(M_1) \cap \overline{\mathcal{L}(P)}) \neq \emptyset$.

[6] Two states are *incompatible* if there exists some string that leads one of them to an accepting state and leads the other to a rejecting state. Otherwise, the two states are *compatible*. The states in a *set of compatible states* are pairwise compatible. A set of compatible states Q is *maximal* if there exists no other set of compatible states Q' such that $Q' \supset Q$.

We then choose the largest set from $\{Q_1, Q_2, Q_3\}$ that contains s_0 as the initial state of the reduced DFA. Here we take Q_3. The next state of Q_3 after reading symbol a is the largest set $Q' \in \{Q_1, Q_2, Q_3\}$ that satisfies $Q' \supseteq \{s' \mid s' = \delta(s, a), \text{ for all } s \in Q_3\} = \{s_0, s_1\}$. Here we get Q_1. Note that we can always find a next state in the reduced DFA. This is because the next states (in the 3DFA) of a set of compatible states are also compatible states. Therefore, the set of the next states (in the 3DFA) is either a set of maximal compatible states or a subset of a set of maximal compatible states. The next states of any

$$\mathcal{C} = (\Sigma, S, s_0, \delta, A, R, D)$$

Fig. 5. The 3DFA to be reduced

$Q \in \{Q_1, Q_2, Q_3\}$ can be found using the same procedure. The procedure terminates after the transition function of the reduced DFA is completely specified. The state Q is an accepting state in the reduced DFA if there exists a state $s \in Q$ such that s is an accepting state in the 3DFA, otherwise it is a rejecting state in the reduced DFA. Formally, we define the reduced DFA $(\Sigma, \hat{S}, \hat{s_0}, \hat{\delta}, \hat{F})$ as follows, let $\mathcal{Q}$ be the sets of maximal compatible states:

- $\hat{S} \subseteq \mathcal{Q}$; $\hat{s_0} = Q \in \mathcal{Q}$, where Q is the largest set that contains s_0;
- $\hat{\delta}(\hat{s}, a) = \hat{s}'$, where $\hat{s}'$ is the largest set $Q \in \mathcal{Q}$ such that $Q \supseteq \{s' \mid s' = \delta(s, a), \text{ for all } s \in \hat{s}\}$;
- $\hat{s} \in \hat{F}$ if there exists a state $s \in \hat{s}$ such that $s \in A$, where A is the set of accepting states in the 3DFA.

According to our experimental results, although the adapted algorithm is not guaranteed to provide an optimal solution, it usually produces a satisfactory one and is much faster than the original version. Besides, since we do not insist on minimality, we also skip *completeness checking* in the adapted version. *Completeness checking* takes a lot of time because the two DFA's C_i^+ and C_i^- can be large and several iteration between *candidate generation* and *completeness checking* may be needed to find a complete 3DFA.

6 Experiments

We evaluated L^{Sep} and its adapted version by two sets of experiments. First, we compared the L^{Sep} algorithm with the algorithm of Gupta *et al.* [14] and that of Grinchtein *et al.* [13] on a large set of randomly-generated sample problems. Second, we evaluated the *adapted L^{Sep}* algorithm and compared it with other automated compositional verification algorithms on the LTSA benchmarks [9]. A more detailed description of the settings of our experiments can be found in [7].

6.1 Experiment 1

We first describe the *sample generator*. Each sample problem has two DFA's $\mathcal{B}_1$ and $\mathcal{B}_2$ such that $\mathcal{L}(\mathcal{B}_1) \subseteq \mathcal{L}(\mathcal{B}_2)$. The *sample generator* has two input
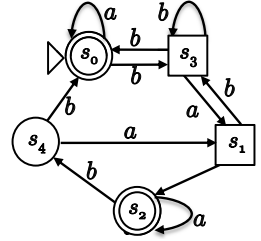
Table 1. Comparison of the Three Algorithms. The row "Avg. DFA Size" is the average size of the two input DFA's $\mathcal{B}_1$ and $\mathcal{B}_2$ in a sample problem. Each column is the average result of 100 sample problems. The row "(i,j)" is the parameters of the sample generator.

Avg. DFA Size	13	21	32	42	54	70	86	102	124
(i,j)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	(10,4)	(11,4)	(12,4)
Algorithms	Average execution time								
L^{Sep}	0.04	0.16	0.4	0.84	1.54	2.5	4.3	6.8	10.9
Gupta [14]	6.6	58.7	266.7	431.5	1308.8	>4000	>4000	>4000	>4000
Grinchtein [13]	51.8	139	255.6	514.7	>4000	>4000	>4000	>4000	>4000
Avg. DFA Size	16	24	36	48	63	80	99	119	142
(i,j)	(4,8)	(5,8)	(6,8)	(7,8)	(8,8)	(9,8)	(10,8)	(11,8)	(12,8)
Algorithms	Average execution time								
L^{Sep}	0.15	0.44	0.96	2.1	3.7	6.4	11	17.8	26.9
Gupta [14]	96.2	625.9	972.3	>4000	>4000	>4000	>4000	>4000	>4000
Grinchtein [13]	813.4	>4000	>4000	>4000	>4000	>4000	>4000	>4000	>4000

Unit: Second

parameters i and j. It first randomly generates[7] two DFA's $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $|\mathcal{A}_1| = |\mathcal{A}_2| = i$. Both use the same alphabet, which is of size j. Then the *sample generator* builds the DFA $\mathcal{B}_1$ by constructing the minimal DFA that recognizes $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{B}_2$ by constructing the minimal DFA that recognizes $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$. The *sample generator* has two important properties: (1) the difference between $|\mathcal{B}_1|$ and $|\mathcal{B}_2|$ is small; (2) there exists a (relatively) small separating DFA for $\mathcal{B}_1$ and $\mathcal{B}_2$.

We used eighteen different input parameters ($i = 4 \sim 12, j = 4, 8$). For each pair (i, j), we randomly generated a set of 100 different sample problems (we eliminated duplications). The *average sizes* of input DFA's ranging from 13 to 142. We also dropped trivial cases ($|\mathcal{B}_1| = 1$ or $|\mathcal{B}_2| = 1$). Table 1 shows the results. We set a timeout of 4000 seconds (for each set of 100 sample problems). If the algorithm did not solve any problem in a set of 100 problems within the timeout period, we mark it as >4000. The time spent on failed tasks is included in the total processing time.

6.2   Experiment 2

We evaluated the adapted L^{Sep} algorithm on the LTSA benchmarks [9]. We compared the adapted L^{Sep} algorithm with the algorithms of Gupta *et al.*, Grinchtein *et al.*, and Cobleigh *et al.* [10]. We implemented all of those algorithms, including the heuristic algorithm for minimizing a 3DFA. We did not consider optimization techniques such as alphabet refinement [6,12]. This is fair because such techniques can also be easily adapted to L^{Sep}. The experimental results are shown in Table 2. The sizes of components are slightly different from the original version because we determinized them. We think the size after determinization can better reflect the difficulty of a benchmark problem. We used the decomposition suggested by the benchmarks to build components M_1 and

---

[7] For each state s in $\mathcal{A}_1$ (respectively $\mathcal{A}_2$) and for each symbol a, a destination state s' in $\mathcal{A}_1$ (respectively $\mathcal{A}_2$) is picked at random and a transition $\delta(s, a) = s'$ is established. Each state has a 50% chance of being selected as a final state.

Table 2. Experimental Results on the LTSA Benchmarks. The "L^{Sep}" column is the result of the adapted L^{Sep} algorithm. "Time" is the execution time in seconds and $|A|$ is the size of the contextual assumption found by the algorithm. "Cobleigh" and "Gupta" give results from [10] and [14], respectively. We highlight in bold font the best results. The column "Problem Size" is the pair $(|M_2|, |M_1| \times |\overline{P}|)$, where $|M_2|$ is the size of the DFA M_2 and $|M_1| \times |\overline{P}|$ is the size of the product of the two DFA's $\overline{M_1}$ and P. The column "MO" is the execution time for monolithic verification. The symbol "-" indicates that the algorithm did not finish within the timeout period. For each row, we use n-m to denote benchmark problem n with m components.

	L^{Sep} Time	—A—	Cobleigh Time	—A—	Gupta Time	—A—	Problem Size	MO		L^{Sep} Time	—A—	Cobleigh Time	—A—	Gupta Time	—A—	Problem Size	MO
1-2	**0.1**	3	170	74	32	3	45, 80	0.08	15-2	**1477**	88	-	-	5992	3	151, 309	0.8
1-3	**0.4**	3	-	-	109	3	82, 848	0.7	15-3	5840	5	-	-	**4006**	3	327, 3369	5.9
1-4	**1.6**	3	-	-	219	3	138, 4046	4.2	15-4	-	-	-	-	**6880**	3	658, 16680	33
2-2	508	7	**89**	52	-	-	39, 89	0.08	19-2	**5.8**	3	-	-	266	3	234, 544	0.3
2-3	-	-	**1010**	93	-	-	423, 142	0.7	19-3	**13**	3	-	-	1392	3	962, 5467	2.9
2-4	-	-	**7063**	152	-	-	2022, 210	4	19-4	**69**	3	-	-	7636	3	**2746, 52852**	35
3-2	**1.9**	3	51	57	140	3	39, 100	0.09	21-3	**45**	3	-	-	4558	3	962, 5394	2.9
3-3	**13**	3	601	110	551	3	423, 164	0.8	21-4	**718**	3	-	-	3839	3	**2746, 51225**	34.8
3-4	**55**	3	4916	189	1639	3	2022, 69	4.2	22-2	**0.6**	3	8	25	12	3	900, 30	0.3
4-2	**5.8**	3	21	35	90	3	39, 87	0.09	22-3	**2.3**	3	1242	193	54	3	7083, 264	4.6
4-3	**20.8**	3	1109	103	433	3	423, 140	0.75	22-4	**11**	3	-	-	170	3	**30936, 2190**	33
4-4	**44.9**	3	6390	156	793	3	2022, 208	4.1	23-2	92	9	**8.9**	37	-	-	50, 40	0.1
5-2	**940**	64	998	127	-	-	45, 133	0.09	24-2	1.2	6	**0.2**	12	1.2	3	13, 14	0.01
7-2	362	39	**48**	46	-	-	39, 104	0.09	24-3	5.1	6	**0.33**	12	-	-	48, 14	0.02
7-3	-	-	**405**	76	-	-	423, 168	0.9	24-4	18	6	**0.63**	12	-	-	157, 14	0.1
7-4	-	-	**3236**	123	-	-	2022, 256	4.1	25-2	**1156**	5	3050	257	-	-	41, 260	0.1
9-2	**1345**	52	4448	240	-	-	45, 251	0.09	26-2	512	38	**239**	121	-	-	65, 123	0.1
10-2	6442	18	-	-	**196**	3	151, 309	0.8	27-2	848	46	**830**	193	-	-	41, 204	0.1
10-3	5347	22	-	-	**601**	3	327, 3369	6.1	28-2	**755**	46	757	185	-	-	41, 188	0.1
10-4	-	-	-	-	**1214**	3	**658, 16680**	33	29-2	926	21	**891**	193	-	-	41, 195	0.1
11-2	**6533**	82	-	-	-	-	151, 515	0.8	30-2	1083	24	**986**	193	-	-	41, 195	0.1
12-2	**36**	4	1654	162	-	-	151, 273	0.8	31-2	**204**	5	274	121	4975	3	65, 165	0.1
12-3	**133**	4	-	-	-	-	327, 2808	6.6	32-2	**9.9**	3	646	193	121	3	41, 261	0.1
12-4	**450**	4	-	-	-	-	**658, 13348**	33	32-3	**44**	3	-	-	-	-	1178, 4806	2.6
									32-4	**886**	3	-	-	-	-	**289, 117511**	382

M_2. Furthermore, we swapped M_1 and M_2; in [9], they check $\mathcal{L}(M_1) \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(M_2) \cap \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(P)$ in the experiments. We swapped them because in the original arrangement, a large portion of the cases have an assumption of size 1. We set a timeout of 10000 seconds. Actually we checked all the 89 LTSA benchmark problems (of 2 ,3, and 4 components). In the table we do not list results with minimal contextual assumption of size 1 (10 cases) and those in which no algorithms finished within the timeout period (30 cases). In addition, we do not list the result of Grinchtein *et al.* because of the space limitation. In this set of experiments, it cannot solve most of the problems within the timeout period (84 cases). Even if it solved the problem (5 cases), it is slower than others.

The adapted L^{Sep} algorithm performs better than all the other algorithms in 30 among the 49 problems. The algorithm of Cobleigh *et al.* wins 14 problems. However, in 8 of the 14 cases (23-2, 24-2, 24-3, 24-4, 26-2, 27-2, 29-2, 30-2), their algorithm finds an assumption with size almost the same as $|\overline{M_1 \times P}|$. In those cases, there is no hope of defeating monolithic verification. In contrast, our algorithm scales better than monolithic verification in several problem sets. For example, in 1-m, 19-m, 22-m, and 32-m, the execution time of the adapted L^{Sep} algorithm grows much slower than monolithic verification. In 1-m and 22-m, we can see that the adapted L^{Sep} algorithm takes more execution time than

monolithic verification when the number of components is 2, but its performance surpasses monolithic verification when the number of components becomes 4.

7   Discussion and Further Work

The algorithm of Gupta *et al.* is *passive*, using only containment queries (which is slightly more general than equivalence queries). From a lower bound result by Angluin [3] on learning with equivalence queries, the query complexity of the algorithm of Gupta *et al.* can be shown to be exponential in the sizes of the minimal DFA's of the two input languages. Moreover, the data structures that they use to represent the samples are essentially trees, which may grow exponentially. These explain why their algorithm does not perform well in the experiments.

The algorithm of Grinchtein *et al.* [13] is an improved version of an earlier algorithm of Pena and Oliveira [16], which is *active.* However, according to our experiments, this improved active algorithm is outperformed by the purely passive learning algorithm of Gupta *et al.* in most cases. The main reason for the inefficiency of this particular active learning algorithm seems to be that the membership queries introduce a lot of redundant samples, even though they reduce the number of iterations required. The redundant samples substantially increase the running time of the exponential procedure of computing the minimal DFA. In contrast, our active algorithm L^{Sep} indeed performs better than the passive algorithm of Gupta *et al.*

The better performance of L^{Sep} can be attributed to the facts that the algorithm utilizes membership queries to accelerate learning and has a more compact representation of the samples (a 3DFA) collected from the queries. For further work, it will be interesting to adapt L^{Sep} for other applications, such as inferring network invariants of parameterized systems and to evaluate the performance of the resulting solutions. Given that L^{Sep} is a better learning algorithm, we hope that other applications will also benefit from it.

References

1. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
3. Angluin, D.: Negative results for equivalence queries. Machine Learning 5(2), 121–150 (1990)
4. Barringer, H., Giannakopoulou, D., Păsăreanu, C.S.: Proof rules for automated compositional verification through learning. In: SAVCBS 2003, pp. 14–21 (2003)
5. Chaki, S., Clarke, E.M., Sinha, N., Thati, P.: Dynamic component substitutability analysis. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 512–528. Springer, Heidelberg (2005)

6. Chaki, S., Strichman, O.: Optimized L*-based assume-guarantee reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007)
7. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating DFA's for compositional verification. Technical Report CMU-CS-09-101, Carnegie Mellon Univeristy (2009)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
9. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Transactions on Software Engineering and Methodology 7(2), 1–52 (2008)
10. 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)
11. Farzan, A., Chen, Y.-F., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Apolloni, B., Howlett, R.J., Jain, L. (eds.) KES 2007, Part II. LNCS, vol. 4693, pp. 2–17. Springer, Heidelberg (2007)
12. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining interface alphabets for compositional verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
13. Grinchtein, O., Leucker, M., Piterman, N.: Inferring network invariants automatically. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 483–497. Springer, Heidelberg (2006)
14. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 420–432. Springer, Heidelberg (2007)
15. Paull, M.C., Unger, S.H.: Minimizing the number of states in incompletely specified sequential switching functions. IRE Transitions on Electronic Computers EC-8, 356–366 (1959)
16. Pena, J.M., Oliveira, A.L.: A new algorithm for the reduction of incompletely specified finite state machines. In: ICCAD 1998, pp. 482–489. ACM Press, New York (1998)
17. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Information and Computation 103(2), 299–347 (1993)
18. Sinha, N., Clarke, E.M.: SAT-based compositional verification using lazy learning. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 39–54. Springer, Heidelberg (2007)

RBAC-PAT: A Policy Analysis Tool for Role Based Access Control[*]

Mikhail I. Gofman[1], Ruiqi Luo[1], Ayla C. Solomon[2], Yingbin Zhang[1], Ping Yang[1], and Scott D. Stoller[3]

[1] Dept. of Computer Science, Binghamton University, NY 13902, USA
[2] Dept. of Computer Science, Wellesley College, Wellesley, MA 02481, USA
[3] Dept. of Computer Science, Stony Brook University, Stony Brook, NY 11794, USA

Abstract. Role-Based Access Control (RBAC) has been widely used for expressing access control policies. Administrative Role-Based Access Control (AR-BAC) specifies how an RBAC policy may be changed by each administrator. Because sequences of changes by different administrators may interact in unintended ways, it is often difficult to fully understand the effect of an ARBAC policy by simple inspection. This paper presents RBAC-PAT, a tool for analyzing RBAC and ARBAC policies, which supports analysis of various properties including reachability, availability, containment, weakest precondition, dead roles, and information flows.

1 Introduction

Role-Based Access Control (RBAC) is widely used for expressing access control policies in areas such as health care and finance. In large organizations, RBAC policies are often managed by multiple administrators with varying authority. An Administrative Role Based Access control (ARBAC) policy specifies how each administrator may change the RBAC policy. Changes by one administrator may interact in unintended ways with changes by other administrators. Consequently, the effect of an ARBAC policy is hard to understand by manual inspection alone.

Policy analysis helps systems designers and administrators understand and debug policies. This paper presents RBAC-PAT, a tool for analyzing various properties of RBAC and ARBAC policies, including (1) *reachability*: e.g., can user u be assigned to role r (called a "goal")? (2) *availability*: e.g., is user u always a member of role r? (3) *role-role containment*: is every member of role r_1 also a member of role r_2? (4) *weakest precondition*: what are the minimal sets of initial roles that enable a user to get added to roles in the goal? (5) *dead roles*: what roles cannot be assigned to any user? and (6) *information flow*: can information flow from object o_1 to object o_2? For properties (1)–(5), the analysis considers all RBAC policies reachable from a given initial RBAC policy by actions allowed by a given ARBAC policy for a given set of administrators.

2   Preliminaries

Role Based Access Control. The central notion of RBAC is that users are assigned to appropriate roles, and roles are granted appropriate permissions. Role hierarchy is a partial order on the set of roles. For example, $GradStudent \succeq Student$ means that role $GradStudent$ is senior to role $Student$, *i.e.*, every member of $GradStudent$ is also implicitly a member of $Student$.

Administrative Role Based Access Control. ARBAC97 [2] controls changes to the user-role assignment, the permission-role assignment and the role hierarchy. Authority to assign users to roles and revoke users from roles are specified by the can_assign and can_revoke relations, respectively. For example, $can\_assign(DeptChair, Grad \wedge \neg RA, TA)$ specifies that the administrative role $DeptChair$ has authority to assign a user who is a member of $Grad$ but not a member of RA to the role TA. A role that appears in a positive precondition, like $Grad$ in this example, is called a *positive role*; similarly, RA is a *negative role* in this example. Authority to assign and revoke permissions is controlled similarly.

3   Tool Description

ARBAC Policy   **Property**

FRONT–END

GUI

Parser/Checker

Parsed Policy

Hierarchy Converter

Converted Policy

Policy Analysis Engine

Evidence Generator

BACK–END

Evidence
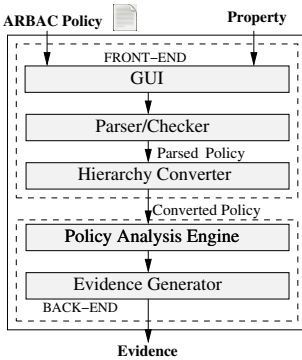
Fig. 1. System architecture

The architecture of RBAC-PAT is shown in Figure 1. Below, we describe its main components.

3.1   Hierarchy Converter

This component converts hierarchical policies into non-hierarchical policies for analysis [3].

3.2   Policy Analysis Engine

Reachability, availability, role-role containment, and weakest precondition. RBAC-PAT implements algorithms we developed for user-role reachability analysis of ARBAC with and without the separate administration restriction [4]. Separate administration requires that administrative roles and regular roles are disjoint. Our algorithms for the other analysis problems are either similar to these algorithms or reduce the problem to user-role reachability analysis [4,3]. We developed forward and backward algorithms for user-role reachability with separate administration and analyzed their parameterized complexity. The idea of parameterized complexity is to identify an aspect of the input that makes the problem computationally hard, introduce a parameter k to measure that aspect, and develop an algorithm that may have high complexity in terms of k, but is polynomial in the overall input size when the value of k is fixed. Such an algorithm is said to be *fixed parameter tractable* with respect to k ("FPT w.r.t. k").

In the forward algorithm, a simple backward slicing transformation eliminates roles and rules irrelevant to the given goal. Next, a *reduced state graph* is constructed; reachability is determined from it. Each node corresponds to an RBAC policy; each edge

corresponds to a change allowed by the ARBAC policy. The following reduction is applied: (1) Transitions that revoke non-negative roles or add non-positive roles are prohibited; (2) Transitions that add non-negative roles or revoke non-positive roles are called *invisible* transitions and get combined with a preceding visible transition to form a single composite transition. The forward algorithm is FPT w.r.t. the number of *mixed roles*, i.e., roles that are both positive and negative. This number is usually significantly smaller than the total number of roles. For example, in ARBAC policies we developed for a university and a health care facility, the percentage of mixed roles is less than 25%.

The backward algorithm has two stages. The first stage uses backward search from the goal to construct a directed graph G. Each node in G is a set of roles, and each edge is labeled with a *can_assign* rule and corresponds to a role assignment action allowed by that rule. However, some negative preconditions of *can_assign* transitions cannot be evaluated during the backward search. The second stage is a forward search that annotates G with the additional information needed to check those preconditions, namely, sets of irrevocable roles that might be left in the state by previous transitions. For ARBAC policies with at most one positive precondition per rule, our backward algorithm is FPT w.r.t. the number of irrevocable roles.

We developed a forward algorithm for analysis of ARBAC without separate administration that is FPT w.r.t. the number of mixed roles and the number of users. We also identified a condition called *hierarchical role assignment* that is often satisfied in practice, and we showed that our algorithms that assume separate administration give accurate results for policies satisfying this condition. Informally, the condition is that an administrator cannot assign users to administrative roles that are not junior to his own administrative role.

ARBAC policy analysis problems could be solved using general-purpose finite-state verification tools, but those tools lack the specialized optimizations in our algorithms and would be asymptotically less efficient for some families of policies. A detailed comparison with related work on verification and security policy analysis appears in [4].

RBAC-PAT computes policy statistics, including the numbers of mixed roles and irrevocable roles, and checks whether separate administration and hierarchical role assignment hold. RBAC-PAT uses this information to try to choose the most appropriate analysis algorithms for a given analysis problem. In cases where separate administration restriction is satisfied and it is unclear whether the forward or the backward algorithm will be faster, RBAC-PAT prompts the user to choose between these algorithms.

Dead role analysis. We developed an algorithm to detect *dead roles* in an ARBAC policy, i.e., roles that cannot be assigned to any user. Dead roles might indicate flaws in the policy. A straightforward algorithm for detecting dead roles is: for every user u_i, compute a set R_i of roles that can be assigned to u_i until all roles have been assigned to some user or all users have been considered; roles not in $\bigcup R_i$ are dead. If the policy satisfies separate administration, the following optimizations are applied: (1) a slicing transformation is used to eliminate roles and rules irrelevant to unassigned roles, and only users with distinct sets of initial roles are considered; and (2) at each step, we consider the user that can potentially be assigned to the most currently unassigned roles.

Information flow analysis. Information flow analysis helps administrators understand the information flows allowed by an RBAC policy. Information can flow directly from

object o_1 to object o_2 if there exists a user that can read from o_1 and write to o_2. Osborn [1] proposed an algorithm for constructing an information flow graph from an RBAC policy, in which an edge $o_1 \rightarrow o_2$ specifies that information can flow directly from o_1 to o_2. We improve this algorithm by eliminating infeasible intermediate edges, for example, edges resulting from roles that have not been assigned to any users. RBAC-PAT also supports information flow queries such as "can information flow, directly or transitively, from object o_1 to object o_2?"

Evidence generation. RBAC-PAT provides evidence that shows why a property holds or is violated. For example, if the answer to a reachability analysis query is yes, RBAC-PAT provides a sequence of administrative actions that leads to the specified role assignment, and highlights the corresponding ARBAC rules in the policy.

3.3   Case Studies

We developed RBAC and ARBAC policies for a university and a health care facility. Here are some sample properties for the university policy: (1) *User-role reachability*: can a user initially in role $DeptChair$ and a user initially in role $Undergrad$ together assign the latter user to $HonorsStudent$? (2) *Weakest Precondition*: what are the weakest preconditions for an administrator initially in $DeptChair$ to assign a user to $HonorsStudent$? (3) *Role-role containment*: is TA contained in $Grad$? (4) *Information flow query*: can information flow from $GradeBook$ to $DeptReport$? RBAC-PAT terminates in at most 0.19 second for all queries we tried. RBAC-PAT also helped uncover some flaws in the original university policy, for example, a place where we accidentally used $Student$ instead of $Undergrad$ and places where we forgot to take role hierarchy into account, *e.g.*, places where we forgot that $Provost$ inherits from $Staff$. Further, in order to validate our FPT results and explore the practical performance of the algorithms, we applied RBAC-PAT to reachability analysis of hundreds of randomly generated policies [4]. For the policies containing 13 reachable mixed roles and 32 roles, RBAC-PAT generates at most 232320 states and 2900920 transitions, and terminates in 8.6 hours. For the policies containing 5 reachable mixed roles and 500 roles, RBAC-PAT generates at most 510 states and 2550 transitions, and terminates in 155 minutes.

References

1. Osborn, S.: Information flow analysis of an RBAC system. In: SACMAT, pp. 163–168 (2002)
2. Sandhu, R., Bhamidipati, V., Munawer, Q.: The ARBAC97 model for role-based administration of roles. TISSEC 2(1), 105–135 (1999)
3. Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.: Policy analysis for administrative role based access control. In: IEEE CSFW, pp. 124–138 (2006)
4. Stoller, S., Yang, P., Ramakrishnan, C.R., Gofman, M.: Efficient policy analysis for administrative role based access control. In: CCS, pp. 445–455 (2007)

ITPN-PerfBound: A Performance Bound Tool for Interval Time Petri Nets

Elina Pacini Naumovich*, Simona Bernardi, and Marco Gribaudo

Dipartimento di Informatica, Università di Torino, Torino (Italy)
{pacini,bernardi,marcog}@di.unito.it

Abstract. The *ITPN-PerfBound* is a tool for the modeling and analysis of Interval Time Petri Nets (ITPN), that is Petri Nets in which firing time intervals, and possibly firing frequency intervals, are associated to transitions. The tool is particularly well-suited in the verification and validation activities of real-time systems, where the main goal is to give guarantees about the worst and best case system performance. The tool has been implemented within the *DrawNET* framework and supports the analysis of ITPN models based on the computation of upper and lower bounds of classical performance metrics, such as throughput and cycle time.

1 Introduction

In the verification and validation activities of real-time systems, one of the main goals is to give guarantees about the best and the worst system performance, e.g., best/worst execution time, before such systems are put into use. Interval Time Petri Nets (ITPNs) and related bound computation techniques can be used for this purpose [1]. ITPNs include Time Petri Nets (TPNs) [2] and TPNs with firing frequency intervals (TPNFs) [1]. TPNs are Petri Nets in which firing time intervals are associated to transitions. TPNFs reduce the non-determinism of free-choice conflicts in TPNs by assigning a firing frequency interval to each transition in free-choice conflict. ITPN performance bound computation techniques are efficient techniques that can be applied to compute bounds of transition throughput, cycle time, and place marking. The bounds are computed by solving a linear programming problem (LPP) derived from the structure of the net, the initial marking and the firing time/frequency interpretation.

Currently, there are many tools for the modeling and analysis of TPNs, while no tools for TPNF are available (e.g., see the Petri Net tool data-base for an updated list [3]). Several of them (e.g., [4,5,6]) provide support for the behavioral analysis of TPNs, based on the use of enumerative techniques. However, to the best of our knowledge, none of them have performance bound analysis capabilities.

In this paper, we present *ITPN-PerfBound*, a graphical tool for the modeling and performance bound analysis of ITPNs. The tool has been implemented within the *DrawNET* framework [7], which provides facilities for the design and solution of models expressed in any graph-based formalism. In particular, *DrawNET* supports a two level structure. The *meta-level* defines the formalism being used, and the *level* describes the model belonging to the formalism. The models can be solved using other existing tools (i.e., GreatSPN [8]), interfaced to the *DrawNET* framework throughout *Solvers*. A Solver is a small interface that translates a model in a particular format (using an entity called *Filter*), and then takes care of invoking the tool that produces the actual results. It also extracts the particular measures and stores them back inside the corresponding model.

Finally, to support the inter-operability between PN tools, *ITPN-PerfBound* includes the possibility of exporting (1) TPN models, by using the standard PNML format [9] enriched with timing information which are compliant with the TINA tool [4], and (2) un-timed PN models toward the GreatSPN tool [8].

2 Overall Architecture and Main Functionalities

ITPN-PerfBound has been implemented within the *DrawNET* framework. We used the *DrawNET* XML-interchange format FDL (*Formalism Definition Language*) for defining the ITPN formalism. The *DrawNET* GUI reads the ITPN formalism, written in FDL and presents to the user a graphical user interface for designing models of that formalism. Figure 1 shows the overall architecture of the *ITPN-PerfBound* tool (part inside the grey rectangle) and its interaction with other Petri Net tools (i.e., TINA and GreatSPN). The components are depicted as rounded-cornered rectangles while the information flow between components is indicated by arrows.

The *DrawNET ITPN module* is the editor component that is used by the modeler to construct ITPN models. The *DrawNET ITPN module* and the *Model/ Result Filters* exchange ITPN models described with the DDL (*Data Definition Language*). The *Model Filter* produces a set of input files for the *ITPN bound solvers*, containing information on the ITPN model definition and on the metrics to be computed. In particular, the un-timed specification of the ITPN model is translated into the format of the GreatSPN tool (.net/.def files), then enabling the user to exploit the GreatSPN facilities for the structural analysis of the un-timed version of such ITPN models. The *Model Filter* translates also the ITPN models into the standard PNML format [9], enriched with timing information which are compliant with the TINA tool.

The *ITPN bound solvers* include a set of solution components, written in C-language, that implement the ITPN performance bound techniques for ITPNs with generic topology and for special structural classes of ITPNs. The *LPP generator* components produce the LPP from the structure of the ITPN model and the initial marking (.net/.def files), the specification of the transition firing time/frequency (.itpn file), the user query about the metric of interest (.cmd file). Two LPP generators have been implemented that create a different LPP,
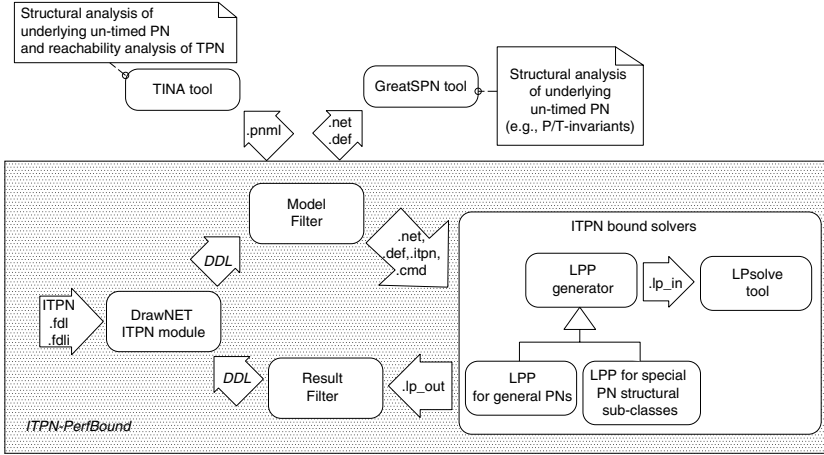
Fig. 1. Components interaction and information flow

according to the type of performance bound technique chosen, and return the LPP written in the input format (.lp_in file) of the free-source *LPsolve tool* [10]. The latter is used to compute the LPP solution (.lp_out file). The results of the performance bound analysis are fed back to the original ITPN model by the *Result Filter* and displayed by *DrawNET ITPN module*.

From the user point of view, the tool provides the following functionalities:

– Create ITPN models by loading the ITPN module from the *DrawNET* GUI and save them in the *DrawNET* XML-interchange format (i.e., mdl).
– Compute performance upper/lower bounds of transition throughput, cycle time and place marking for ITPN model with generic topology. The result of the analysis (i.e., the optimal value of the LPP objective function) is displayed in the *Properties* tab associated to the corresponding net object (transition/place) in the *DrawNET ITPN module*.
– Compute best/worst case performance transition cycle time and throughput for special structural classes of ITPNs (i.e., Marked Graphs and 1-consistent monoT-semiflows). The presentation of the results (i.e., the LPP optimal solution and the corresponding value of the objective function) is both textual and graphical: the bound values are displayed in the *Properties* tab associated to the net model in the *DrawNET ITPN module* and the slowest sub-net of the ITPN model (i.e., the sub-net with the highest cycle time) is highlighted in red.
– Export the ITPN models toward the GreatSPN tool [8] and the standard PNML format, enriched with timing information which are compliant with the TINA tool [4].

3   Applications and Conclusion

The *ITPN-PerfBound* tool has been applied to several case studies and examples from the literature. Among them, we can mention: a flexible manufacturing system [11] that produces short-stroke cylinders; the alternating bit protocol, modeled with an un-timed PN in [12]; and a computer-assisted braking system for vehicles [13]. The integration of the ITPN bound solvers within the *DrawNET* environment, well as the implementation of the filters towards GreatSPN and TINA, has been considerably reduced the time devoted to the V&V activities of the considered case studies. New GUI facilities are going to be implemented that allows the analyst to visualize the net labels, parameters, timing specifications and results in the *DrawNET* model job window and to export the net to the SVG format. Possible future developments concern the implementation of new filters that enable to import, in the *ITPN-PerfBound* tool, the ITPN models specified either in the PNML standard format or in the GreatSPN format.

The tool is available at the following URL:

<div align="center">http://www.draw-net.com/ITPN-PerfBound.</div>

References

1. Bernardi, S., Campos, J.: On Performance Bounds for Interval Time Petri Nets. In: Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST'04), Enschede, The Netherlands, pp. 50–59. IEEE Computer Society Press, Los Alamitos (2004)
2. Merlin, P., Faber, D.: Recoverability of communication protocols. IEEE Trans Commun. COM-24(9) (1976)
3. PetriNetsWorld: Petri nets tool database,
   `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`
4. Berthomieu, B.: The TIme petri Net Analyzer (TINA) toolbox,
   `http://www.laas.fr/tina/`
5. Roux, O., et al.: The Roméo tool, `http://romeo.rts-software.org`
6. Vicario, E., et al.: The ORIS tool,
   `http://www.stlab.dsi.unifi.it/oris/index.html`
7. Gribaudo, M., Codetta Raiteri, D.G.F.: The DrawNET Modelling System: a framework for the design and the solution of single-formalism and multi-formalis models. Technical Report TR-INF-2006-01-UNIPMN (January 2006)
8. PerfGroup: The GreatSPN tool, `http://www.di.unito.it/~greatspn`
9. Billington, J., et al.: The Petri Net Markup Language. Standard ISO/IEC-15909-2
10. Berkelaar, M., et al.: LP_SOLVE: library for solving linear (integer) programming problems, `http://lpsolve.sourceforge.net/5.5/`
11. Paoli, A., Sartini, M., Tilli, A.: Rapid prototyping of logic control in industrial automation exploiting the generalized actuator approach. In: Proc. of 13th IEEE International Conference on Emerging Technologies and Factory Automation, Hamburg, Germany. IEEE, Los Alamitos (2008)
12. Diaz, M., Azema, P.: Petri net based models for the specification and validation of protocols. In: Rozenberg, G. (ed.) APN 1984. LNCS, vol. 188, pp. 101–121. Springer, Heidelberg (1985)
13. Bernardi, S., Campos, J., Merseguer, J.: Timing-failure risk assessment based on Petri net bounding techniques. Technical report, University of Torino, Italy (June 2008)

Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches

Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez

IRCCyN, CNRS UMR 6597, Nantes, France
{Didier.Lime,Olivier-h.Roux,Charlotte.Seidner,
Louis-Marie.Traonouez}@irccyn.ec-nantes.fr

Abstract. Last time we reported on Romeo, analyses with this tool were mostly based on translations to other tools. This new version provides an integrated TCTL model-checker and has gained in expressivity with the addition of parameters. Although there exists other tools to compute the state-space of stopwatch models, Romeo is the first one that performs TCTL model-checking on stopwatch models. Moreover, it is the first tool that performs TCTL model-checking on timed parametric models. Indeed, Romeo now features an efficient model-checking of time Petri nets using the Uppaal DBM Library, the model-checking of stopwatch Petri nets and parametric stopwatch Petri nets using the Parma Polyhedra Library and a graphical editor and simulator of these models. Furthermore, its audience has increased leading to several industrial contracts. This paper reports on these recent developments of Romeo.

Keywords: Time Petri nets, model-checking, stopwatches, parameters, TCTL, tool.

1 Introduction

Time Petri nets (*TPNs*) [1] are a classical time extension of Petri nets. They allow an easy representation of real-time systems features such as synchronization and parallelism. State reachability is decidable for bounded *TPNs*, which is sufficient for virtually all practical purposes.

However, it is also often useful to model actions that can be suspended and later resumed. Several extensions of time Petri nets have been proposed to express the preemptive scheduling of tasks, such as *Scheduling-TPNs* [2], inhibitor hyperarc *TPNs* (*ITPNs*) [3] or *Preemptive-TPNs* [4]. All these models belong to the class of *TPNs* extended with stopwatches (*SwPNs*)[5]. Reachability and most other properties of interest are however undecidable for *SwPNs*, even when bounded [5].

Furthermore, the design of a system often benefits from the use of parameters, e.g. when specifications are not yet completely defined. Parametric *TPNs* (*PTPNs*) and parametric *SwPNs* (*PSwPNs*) are parametric extensions of *TPNs* and *SwPNs* that can be used to perform parametric model-checking [6]. The goal is to synthesize constraints on the parameters which helps the system design.

2   Presentation of Romeo

The ROMEO tool [1] (available for Linux, MacOSX and Windows platforms) consists of a graphical user interface (GUI) (written in Tcl/Tk), to edit and simulate *TPNs*, and a computation module MERCUTIO (written in C++), that performs model-checking and state-space computation.

The two other main tools for the analysis of *TPNs* and *SwPNs* are TINA [7] and ORIS [8]. TINA has many interesting features, including the computation of a graph preserving CTL (Computation Tree Logic) properties and an off-line full LTL model-checker. ORIS also has several unique features of interest among which the most notable is probably the analysis of time Petri nets with stochastic aspects.

Design and Simulation. The GUI allows Petri nets edition and features three different modes for each supported extension (*TPNs*, *Scheduling-TPNs* or *ITPNs*). In each mode, a parametric extension is available (*PTPNs*, *Scheduling-PTPNs* or *PITPNs*). In these latter extensions, ROMEO supports the use of parametric linear expressions in the time bounds of the transitions, and allows to add linear constraints on the parameters to restrict their domain.

The GUI also features an on-line interactive simulator for scenario testing. It allows to study a particular trace in the state-space of the model, either with the state-class method [9] or with the zone-based method [10] (only for *TPNs*).

On-the-Fly Model-Checking. Through the computation module MERCUTIO, ROMEO can perform model-checking of time Petri nets models for quantitative temporal logic formulae (TCTL) [11]. We consider a restricted subset of TCTL formulae with no recursion in the formulae for which we can propose an efficient on-the-fly model-checking. Moreover, this subset appears to be sufficient to verify many interesting properties on time models. Reachability properties can be checked with formulae such as $\exists\Diamond_{[a,b]}(p)$ (where $[a,b]$ is a time interval, with b possibly infinite, and p a property on the markings of the net) and safety properties with $\forall\Box_{[a,b]}(p)$. Liveness properties can be checked with $\forall\Diamond_{[a,b]}(p)$ or by using a bounded response property such as $p \leadsto_{[0,b]} q$. It is equivalent to $\forall\Box(p \Rightarrow \forall\Diamond_{[0,b]}(q))$, and thus allows one level of recursion.

This subset allows to implement efficient model-checking algorithms for *TPNs* with the state-class graph [12]. In bounded *TPNs*, the algorithm is based on DBMs (Difference Bounds Matrix) and the implementation uses the Uppaal DBM Library [13] to encode the firing domains. For *SwPNs* and parametric *SwPNs*, the reachability problem is undecidable and as a consequence semi-algorithms are implemented in ROMEO. In these models, firing domains are encoded with polyhedra by using the Parma Polyhedra Library [14].

With parametric nets, ROMEO can verify parametric TCTL formulae in which the bounds of the temporal constraints (a and b in the above examples) can be replaced by parameters. The goal is to determine the valuations of these parameters, such that for these valuations the model verifies the formula. The

---

[1] Download at: `http://romeo.rts-software.org/`

semi-algorithms implemented are based on the parametric state-class graph [6]. As a result, Romeo synthesizes a set of constraints (a disjunction of polyhedra, also encoded with the Parma Polyhedra Library) to represent the set of these valuations.

State-Space Computation. Romeo implements two state-space computation methods, the state-class graph and the zone-based graph.

Romeo computes the state-class graph (SCG) that preserves LTL properties [9]. The algorithm is based on DBMs for bounded *TPNs* and the semi-algorithm is based on polyhedra for *PTPNs*. For *SwPNs*, exact computation semi-algorithms are implemented by using either polyhedra or both polyhedra and DBMs. Finally an overapproximating semi-algorithm is also available for *SwPNs* and uses DBMs.

For *TPNs*, Romeo generates a zone-based graph [10] that preserves markings and converges by inclusion.

Finally, translations from *TPNs* to Timed Automata are available through the state-class automaton or by a structural translation. *SwPNs* can be translated into Stopwatch Automata by an overapproximation method. Both were accounted for in [15].

3   New Functionalities

Since the last paper about Romeo [15], several enhancements have been made to the tool.

Regarding expressiveness, we have added the support for several classical special arcs: read arcs (resp. logical inhibitor arcs) test the number of tokens in a place, without consuming them, for the relation "greater than" (resp. "less than"). Reset arcs empty a place of all its token (regardless of their number) after the firing of a transition.

Furthermore, in addition to the scheduling extension, stopwatches can now be defined using time inhibitor arcs [3].

Regarding verification, we now have a model-checker for TCTL properties for *TPNs*, *SwPNs* and their parametric counterparts. It allows to work with Romeo without the help of other tools.

This is, to our knowledge, the first TCTL model-checker for (bounded) time Petri nets but also the first one for stopwatch models.

Furthermore, as already mentioned, Romeo now implements a new framework for the design and verification of parametric time Petri nets as described in [6]. The associated parametric model-checker is also, to our knowledge, the first one to perform TCTL parametric model-checking on timed parametric models.

4   Conclusion

Romeo is one of the three main tools on time and stopwatch Petri nets. It performs state-space computation and simulation based on both the state-class

method and the zone-based method. Moreover, it performs on-the-fly TCTL model-checking of time, stopwatch and parametric Petri nets. Its implementation is very efficient thanks to the use of two robust libraries, namely the Uppaal DBM Library and the Parma Polyhedra Library.

ROMEO has many industrial users such as *DGA*, *SODIUS*, *Dassault Aviation* and *EADS* leading to several industrial contracts and partnerships.

References

1. Merlin, P.: A study of the recoverability of computing systems. PhD thesis, Department of Information and Computer Science, Univ. of California, Irvine (1974)
2. Roux, O., Déplanche, A.M.: A t-time Petri net extension for real time-task scheduling modeling. European Journal of Automation (JESA) 36(7), 973–987 (2002)
3. Roux, O.H., Lime, D.: Time petri nets with inhibitor hyperarcs. Formal semantics and state space computation. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 371–390. Springer, Heidelberg (2004)
4. Bucci, G., Fedeli, A., Sassoli, L., Vicario, E.: Time state space analysis of real-time preemptive systems. IEEE trans. on Soft. Eng. 30(2), 97–111 (2004)
5. Berthomieu, B., Lime, D., Roux, O.H., Vernadat, F.: Reachability problems and abstract state spaces for time Petri nets with stopwatches. Discrete Event Dynamic Systems 17(2), 133–158 (2007)
6. Traonouez, L.-M., Lime, D., Roux, O.H.: Parametric model-checking of time Petri nets with stopwatches using the state-class graph. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 280–294. Springer, Heidelberg (2008)
7. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool tina – construction of abstract state spaces for Petri nets and time Petri nets. Int. Journal of Production Research 42(4) (July 2004), `http://www.laas.fr/tina/`
8. Bucci, G., Sassoli, L., Vicario, E.: Oris: A tool for state-space analysis of real-time preemptive systems. In: QEST 2004 (2004)
9. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE trans. on Soft. Eng. 17(3), 259–273 (1991)
10. Gardey, G., Roux, O.H., Roux, O.F.: State space computation and analysis of time Petri nets. Theory and Practice of Logic Programming (TPLP) 6(3), 301–320 (2006); Copyright Cambridge Press
11. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and Computation 104, 2–34 (1993)
12. Hadjidj, R., Boucheneb, H.: On-the-fly TCTL model checking for time Petri nets using state class graphs. In: ACSD, pp. 111–122. IEEE Computer Society Press, Los Alamitos (2006)
13. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1–2), 134–152 (1997)
14. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy (2006)
15. Gardey, G., Lime, D., Magnin, M., Roux, O(H).: Roméo: A tool for analyzing time petri nets. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005)

Alpaga: A Tool for Solving Parity Games with Imperfect Information

Dietmar Berwanger[1], Krishnendu Chatterjee[2], Martin De Wulf[3],
Laurent Doyen[3,4], and Thomas A. Henzinger[4]

[1] LSV, ENS Cachan and CNRS, France
[2] CE, University of California, Santa Cruz, USA
[3] Université Libre de Bruxelles (ULB), Belgium
[4] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. Alpaga is a solver for two-player parity games with imperfect information. Given the description of a game, it determines whether the first player can ensure to win and, if so, it constructs a winning strategy. The tool provides a symbolic implementation of a recent algorithm based on antichains.

1 Introduction

Alpaga is a tool for solving parity games with imperfect information. These are turn-based games played on a graph by two players, one of them having imperfect information about the current state of the play. We consider objectives over infinite paths specified by parity conditions that can express safety, reachability, liveness, fairness, and most properties commonly used in verification. Given the description of a game, the tool determines whether the imperfect information player has a winning strategy for the parity objective and, if this is the case, it constructs such a winning strategy.

The Alpaga implementation is based on a recent technique using *antichains* for solving games with imperfect information efficiently [3], and for representing the strategies compactly [2]. To the best of our knowledge, this is the first implementation of a tool for solving parity games with imperfect information.

In this paper, we outline the antichain technique which is based on fixed-point computations using a compact representation of sets. Our algorithm essentially iterates a *controllable predecessor* operator that returns the states from which a player can force the play into a given target set in one round. For computing this operator, no polynomial algorithms is known. We propose a new symbolic implementation based on BDDs to avoid a naive enumerative procedure.

Imperfect-information games arise in several key applications related to verification and synthesis of reactive systems, such as (a) synthesis of controllers for plants with unobservable transitions; (b) distributed synthesis of processes with private variables not visible to other processes; (c) synthesis of robust controllers; (d) synthesis of automata specifications where only observations of automata are visible, and (e) the decision and simulation problem of quantitative specification languages; (f) model-checking secrecy and information flow. We believe that the tool Alpaga will make imperfect information games a useful framework for designers in the above applications.

An example of distributed-system synthesis has been solved with the tool. We have considered the design of a mutual-exclusion protocol for two processes. The tool Alpaga was able to synthesize a winning strategy for a requirement of mutual exclusion and starvation freedom which corresponds to Peterson's protocol. Details can be found in an extended version of this paper [1].

2 Games and Algorithms

Let Σ be a finite alphabet of actions and let Γ be a finite alphabet of observations. A *game structure with imperfect information* over Σ and Γ is a tuple $G = (L, l_0, \Delta, \gamma)$, where

- L is a finite set of locations (or states), $l_0 \in L$ is the initial location;
- $\Delta \subseteq L \times \Sigma \times L$ is a set of labelled transitions such that for all $\ell \in L$ and all $a \in \Sigma$, there exists $\ell' \in L$ such that $(\ell, a, \ell') \in \Delta$, i.e., the transition relation is total;
- $\gamma : \Gamma \to 2^L \setminus \emptyset$ is an observability function that maps each observation to a set of locations such that the set $\{\gamma(o) \mid o \in \Gamma\}$ partitions L. For each $\ell \in L$, let $\mathsf{obs}(\ell) = o$ be the unique observation such that $\ell \in \gamma(o)$.

The game on G is played in rounds. Initially, a token is placed in location l_0. In every round, Player 1 first chooses an action $a \in \Sigma$, and then Player 2 moves the token to an a-successor ℓ' of the current location ℓ, i.e., such that $(\ell, a, \ell') \in \Delta$. Player 1 does not see the current location ℓ of the token, but only the observation $\mathsf{obs}(\ell)$ associated to it. A *strategy* for Player 1 in G is a function $\alpha : \Gamma^+ \to \Sigma$. The set of possible *outcomes* of α in G is the set $\mathsf{Outcome}(G, \alpha)$ of sequences $\pi = \ell_1 \ell_2 \ldots$ such that $\ell_1 = l_0$ and $(\ell_i, \alpha(\mathsf{obs}(\ell_1 \ldots \ell_i)), \ell_{i+1}) \in \Delta$ for all $i \geq 1$. A *visible parity condition* on G is defined by a function $p : \Gamma \to \mathbb{N}$ that maps each observation to a non-negative integer priority. We say that a strategy α for Player 1 is *winning* if for all $\pi \in \mathsf{Outcome}(G, \alpha)$, the least priority that appears infinitely often in π is even.

To decide whether Player 1 is winning in a game G, the basic approach consists in tracing the *knowledge* of Player 1, represented a set of locations called a *cell*. The initial knowledge is the cell $s_0 = \{l_0\}$. After each round, the knowledge s of Player 1 is updated according to the action a she played and the observation o she receives, to $s' = \mathsf{post}_a(s) \cap \gamma(o)$ where $\mathsf{post}_a(s) = \{\ell' \in L \mid \exists \ell \in s : (\ell, a, \ell') \in \Delta\}$.

Antichain algorithm. The antichain algorithm is based on the *controllable predecessor* operator $\mathsf{CPre} : 2^S \to 2^S$ which, given a set of cells q, computes the set of cells q' from which Player 1 can force the game into a cell of q in one round:

$$\mathsf{CPre}(q) = \{s \subseteq L \mid \exists a \in \Sigma \cdot \forall o \in \Gamma : \mathsf{post}_a(s) \cap \gamma(o) \in q\}. \tag{1}$$

The key of the algorithm relies on the fact that $\mathsf{CPre}(\cdot)$ preserves downward-closedness. A set q of cells is *downward-closed* if, for all $s \in q$, every subset $s' \subseteq s$ is also in q. Downward-closed sets q can be represented succinctly by their maximal elements $r = \lceil q \rceil = \{s \in q \mid \forall s' \in q : s \not\subset s'\}$, which form an *antichain*. With this representation, the controllable predecessor operator is defined by

$$\mathsf{CPre}(r) = \lceil \{s \subseteq L \mid \exists a \in \Sigma \cdot \forall o \in \Gamma \cdot \exists s' \in r : \mathsf{post}_a(s) \cap \gamma(o) \subseteq s'\} \rceil. \tag{2}$$

Strategy construction. The implementation of the strategy construction is based on [2]. The algorithm of [2] employs antichains to compute winning strategies for imperfect-information parity games in an efficient and compact way: the procedure is similar to the classical algorithm of McNaughton [4] and Zielonka [5] for perfect-information parity games, but, to preserve downwards closure, it avoids the complementation operation of the classical algorithms by recurring into subgames with an objective obtained as a boolean combination of reachability, safety, and reduced parity objectives.

Strategy simplification. A strategy in a game with imperfect information can be represented by a set $\Pi = \{(s_1, \mathsf{rank}_1, a_1), \ldots, (s_n, \mathsf{rank}_n, a_n)\}$ of triples $(s_i, \mathsf{rank}_i, a_i) \in 2^L \times \mathbb{N} \times \Sigma$ where s_i is a cell, and a_i is an action. Such a triple assigns action a_i to every cell $s \subseteq s_i$; since a cell s may be contained in many s_i, we take the triple with minimal value of rank_i. Formally, given the current knowledge s of Player 1, let $(s_i, \mathsf{rank}_i, a_i)$ be a triple with minimal rank in Π such that $s \subseteq s_i$ (such a triple exists if s is a winning cell); the strategy represented by Π plays the action a_i in s.

Our implementation applies the following rules to simplify the strategies and obtain a compact representation of winning strategies in parity games with imperfect information.

(*Rule* 1) In a strategy Π, retain only elements that are maximal with respect to the following order: $(s, \mathsf{rank}, a) \succeq (s', \mathsf{rank}', a')$ if $\mathsf{rank} \leq \mathsf{rank}'$ and $s' \subseteq s$. Intuitively, the rule specifies that we can delete (s', rank', a') whenever all cells contained in s' are also contained in s; since $\mathsf{rank} \leq \mathsf{rank}'$, the strategy can always choose (s, rank, a) and play a.

(*Rule* 2) In a strategy Π, delete all triples $(s_i, \mathsf{rank}_i, a_i)$ such that there exists $(s_j, \mathsf{rank}_j, a_j) \in \Pi$ ($i \neq j$) with $a_i = a_j$, $s_i \subseteq s_j$ (and hence $\mathsf{rank}_i < \mathsf{rank}_j$ by Rule 1), such that for all $(s_k, \mathsf{rank}_k, a_k) \in \Pi$, if $\mathsf{rank}_i \leq \mathsf{rank}_k < \mathsf{rank}_j$ and $s_i \cap s_k \neq \emptyset$, then $a_i = a_k$. Intuitively, the rule specifies that we can delete $(s_i, \mathsf{rank}_i, a_i)$ whenever all cells contained in s_i are also contained in s_j, and the action a_j is the same as the action a_i. Moreover, if a cell $s \subseteq s_i$ is also contained in s_k with $\mathsf{rank}_i \leq \mathsf{rank}_k < \mathsf{rank}_j$, then the action played by the strategy is also $a_k = a_i = a_j$.

3   Implementation

Computing $\mathsf{CPre}(\cdot)$ is likely to require time exponential in the number of observations (a natural decision problem involving $\mathsf{CPre}(\cdot)$ is NP-hard [2]). Therefore, it is natural to let the BDD machinery evaluate the universal quantification over observations in (2). We present a BDD-based algorithm to compute $\mathsf{CPre}(\cdot)$.

Let $L = \{\ell_1, \ldots, \ell_n\}$ be the state space of the game G. A cell $s \subseteq L$ can be represented by a valuation v of the boolean variables $\bar{x} = x_1, \ldots, x_n$ such that, for all $1 \leq i \leq n$, $\ell_i \in s$ iff $v(x_i) = \mathrm{true}$. A BDD over $x_1, \ldots, x_n$ is called a *linear encoding*, it encodes a set of cells. A cell $s \subseteq L$ can also be represented by a BDD over boolean variables $\bar{y} = y_1, \ldots, y_m$ with $m = \lceil \log_2 n \rceil$. This is called a *logarithmic encoding*, it encodes a single cell.

We represent the transition relation of G by the $n \cdot |\Sigma|$ BDDs $T_a(\ell_i)$ ($a \in \Sigma, 1 \leq i \leq n$) with logarithmic encoding over $\bar{y}$. So, $T_a(\ell_i)$ represents the set $\{\ell_j \mid (\ell_i, a, \ell_j) \in$

$\Delta\}$. The observations $\Gamma = \{o_1, \ldots, o_p\}$ are encoded by $\lceil \log_2 p \rceil$ boolean variables $b_0, b_1, \ldots$ in the BDD B_Γ defined by

$$B_\Gamma \equiv \bigwedge_{0 \le j \le p-1} \bar{b} = [j]_2 \to C_{j+1}(\bar{y}),$$

where $[j]_2$ is the binary encoding of j and $C_1, \ldots, C_p$ are BDDs that represent the sets $\gamma(o_1), \ldots, \gamma(o_p)$ in logarithmic encoding.

Given the antichain $q = \{s_1, \ldots, s_t\}$, let S_k ($1 \le k \le t$) be the BDDs that encode the set s_k in logarithmic encoding over $\bar{y}$. For each $a \in \Sigma$, we compute the BDD CP_a in linear encoding over $\bar{x}$ as follows:

$$\mathsf{CP}_a \equiv \forall \bar{b} \cdot \bigvee_{1 \le k \le t} \bigwedge_{1 \le i \le n} x_i \to \left[\forall \bar{y} \cdot \left(T_a(\ell_i) \wedge B_\Gamma \right) \to S_k \right].$$

Then, we define $\mathsf{CP} \equiv \bigvee_{a \in \Sigma} \mathsf{CP}_a(q)$, and we extract the maximal elements in $\mathsf{CP}(\bar{x})$ as follows, with ω a BDD that encodes the relation of (strict) set inclusion $\subset$:

$$\omega(\bar{x}, \bar{x}') \equiv \left(\bigwedge_{i=1}^{n} x_i \to x_i' \right) \wedge \left(\bigvee_{i=1}^{n} x_i \ne x_i' \right),$$

$$\mathsf{CP}^{\min}(\bar{x}) \equiv \mathsf{CP}(\bar{x}) \wedge \neg \exists \bar{x}' \cdot \omega(\bar{x}, \bar{x}') \wedge \mathsf{CP}(\bar{x}').$$

Finally, we construct the antichain $\mathsf{CPre}(q)$ as the following set of BDDs in logarithmic encoding: $\mathsf{CPre}(q) = \{s \mid \exists v \in \mathsf{CP}^{\min} : s = \{\ell_i \mid v(x_i) = \mathsf{true}\}\}$.

Features of the tool. The input of the tool is a file describing the transitions and observations of the game graph. The output is the set of maximal winning cells, and a winning strategy in compact representation. We have also implemented a simulator to let the user play against the strategy computed by the tool. The user has to provide an observation in each round (or may let the tool choose one randomly). The web page of the tool is `http://www.antichains.be/alpaga`. We provide the source code, the executable, an online demo, and several examples.

References

1. Berwanger, D., Chatterjee, K., De Wulf, M., Doyen, L., Henzinger, T.A.: Alpaga: A tool for solving parity games with imperfect information. Technical Report MTC-REPORT-2008-007, EPFL (2008), `http://infoscience.epfl.ch/record/130681`
2. Berwanger, D., Chatterjee, K., Doyen, L., Henzinger, T.A., Raje, S.: Strategy construction for parity games with imperfect information. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 325–339. Springer, Heidelberg (2008)
3. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games of incomplete information. Logical Methods in Computer Science 3(3:4) (2007)
4. McNaughton, R.: Infinite games played on finite graphs. Annals of Pure and Applied Logic 65(2), 149–184 (1993)
5. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200, 135–183 (1998)

Compositional Predicate Abstraction from Game Semantics[*]

Adam Bakewell and Dan R. Ghica

University of Birmingham, UK

Abstract. We introduce a technique for using conventional predicate abstraction methods to reduce the state-space of models produced using game semantics. We focus on an expressive procedural language that has both local store and local control, a language which enjoys a simple game-semantic model yet is expressive enough to allow non-trivial examples. Our compositional approach allows the verification of incomplete programs (e.g. libraries) and offers the opportunity for new heuristics for improved efficiency. Game-semantic predicate abstraction can be embedded in an abstraction-refinement cycle in a standard way, resulting in an improved version of our experimental model-checking tool MAGE, and we illustrate it with several toy examples.

1 Introduction

The most important technical challenge for automatic software verification is the so-called *state-explosion problem*, the fact that the state-complexity of the model checking problem is exponential in the size of the program. As a direct consequence of this, automatic verification is said not to *scale*, i.e. only rather small programs can be handled.

A variety of techniques are used to handle the systems with very large state spaces that occur in automatic verification. Taken together, they can lead to surprisingly effective tools, which can handle fully automatically an impressive range of programs [1,2]. But in a series of papers [3,4,5,6] we have argued that while such techniques are very effective on small to medium sized programs, in order for automatic verification to scale up to large and very large programs it is necessary to be based on *compositional* methods, i.e. have the ability to verify *fragments* of programs, then make correctness judgements about the whole based on correctness judgements about the parts. We believe game semantics [7,8] provides a solid theoretical foundation on which such methods can be developed.

In this paper we develop a *predicate abstraction* [9] from game-based models. The technical challenge is combining the compositional and semantic-directed model construction of game semantics with the syntactic constructs of predicate abstraction and its essential use of global state. In the paper we formulate predicate abstraction for games, prove relevant technical results (decidability, soundness of approximation), discuss new heuristics stemming from this style of

---

predicate abstraction and illustrate it with some examples. The implementation is based on our existing experimental tool Mage.[1]

2   The Language

The technique that we present here can be used to abstract any programs written in a language that has a game-semantic model. To have a focused presentation we will select a fragment of the language that is expressive enough to allow interesting examples, yet simple enough to allow a concise presentation. We call this language IAL. The starting point is IA [10], a well studied language which combines lambda calculus with the simple imperative language. We will use an enhanced variant of a language that in addition to local variables also uses block-structured control, a generalisation of C's `break` and `continue` operations. A similar language, IAX, was studied by Laird [11]. IA-like languages are supposed to use lambda-abstraction uniformly over all types, but this, in conjunction with the call-by-name procedural mechanism, leads to confusing phenomena such as *interference* or *bad variables* [12]. To avoid such issues, which raise the complexity of our presentation but are ultimately irrelevant to the matter of predicate abstraction, we impose some restrictions on the way variables and labels can be used in the language by disallowing variable and label-typed terms in the language. Variables and labels are "named constants" rather than programming language identifiers [13]. We disallow recursion and higher-order functions because they introduce infinite-state models in a way that is not related to the store. Finally to further focus the presentation on store abstraction rather than functional aspects, we only allow a very simple function-definition mechanism, similar to that of C, where all functions are defined in global scope. This language is in general quite close to a large subset of C and we are building up towards real code in the near future.

2.1   Syntax and Operational Semantics

IAL has a discrete set of labels $\mathcal{L}$ and a discrete set of locations $\mathcal{C}$. The base types T of the language are commands com, booleans bool and integers int. Function types are defined by the grammar $U ::= T_1 \times \cdots \times T_k \to T$. For each type T there is a discrete set of identifiers of that type $\mathcal{F}_T$. We use a distinct type prog for programs. The type rules of the language are given in Fig. 1, where by $\mathcal{L}(M)$ and $\mathcal{C}(M)$ we mean the set of labels and locations, respectively, used in M and by $\mathcal{V}(M)$ the set of (free) variables of a term.

The rules for new, break, continue perform the introduction of a fresh location or label name x. This is apparently syntactically restrictive, e.g. the term new x.new x.!x does not type-check, but any such term can be alpha-converted to a legal term, i.e. new x.new y.!y.

The "big-step" operational semantics are standard for an IA-like language. Let $\mathbf{V}$ be the set of values, including natural numbers, booleans and skip, $\mathbf{V} =$

---

[1] `http://www.cs.bham.ac.uk/research/projects/mage`

$$\frac{x \in \mathcal{F}_T}{x : T} \qquad n : \texttt{int} \qquad \texttt{true} : \texttt{bool} \qquad \texttt{false} : \texttt{bool} \qquad \texttt{skip} : \texttt{com}$$

$$\frac{M : \texttt{com} \qquad N : T}{M;N : T} \qquad \frac{M : \texttt{int} \qquad N : \texttt{int}}{M \oplus N : \texttt{int}} \qquad \frac{B : \texttt{bool} \qquad M_i : T}{\texttt{if } B \texttt{ then } M_1 \texttt{ else } M_2 : T}$$

$$\frac{x \in \mathcal{C} \qquad M : \texttt{int}}{x\texttt{:=}M : \texttt{com}} \qquad \frac{x \in \mathcal{C}}{!x : \texttt{int}} \qquad \frac{x \in \mathcal{L}}{\texttt{goto } x : \texttt{com}} \qquad \frac{f \in \mathcal{F}_{T_1 \times \cdots \times T_k \to T} \qquad M_i : T_i}{f(M_1,\ldots,M_k) : T}$$

$$\frac{M : T \qquad x \notin \mathcal{C} \setminus \mathcal{C}(M)}{\texttt{new } x.M : T} \qquad \frac{M : \texttt{com} \qquad x \notin \mathcal{L} \setminus \mathcal{L}(M)}{\texttt{break } x.M : \texttt{com}} \qquad \frac{M : \texttt{com} \qquad x \notin \mathcal{L} \setminus \mathcal{L}(\mathcal{M})}{\texttt{cont } x.M : \texttt{com}}$$

$$\frac{M : T}{M : \texttt{prog}} \qquad \frac{\mathcal{V}(M) = \{x_1,\ldots,x_k\} \qquad f \in \mathcal{F}_{T_1 \times \cdots \times T_k \to T} \qquad M : T \qquad N : \texttt{prog}}{\texttt{let } f(x_1,\ldots,x_k) = M \texttt{ in } N : \texttt{prog}}$$

Fig. 1. Typing rules for IAL

$$\frac{M, \Sigma \Downarrow \texttt{skip}, \Sigma' \qquad N, \Sigma' \Downarrow E, \Sigma''}{M;N, \Sigma \Downarrow E, \Sigma''} \qquad \frac{M, \Sigma \Downarrow G, \Sigma'}{M;N, \Sigma \Downarrow G, \Sigma''}$$

$$\frac{M, \Sigma \Downarrow m, \Sigma' \qquad N, \Sigma' \Downarrow G, \Sigma''}{M \oplus N, \Sigma \Downarrow G, \Sigma''} \qquad \frac{M, \Sigma \Downarrow G, \Sigma'}{M \oplus N, \Sigma \Downarrow G, \Sigma''}$$

$$\frac{M, \Sigma \Downarrow m, \Sigma' \qquad N, \Sigma' \Downarrow n, \Sigma'' \qquad p = m \oplus n}{M \oplus N, \Sigma \Downarrow p, \Sigma''}$$

$$\frac{B, \Sigma \Downarrow b, \Sigma' \qquad M_b, \Sigma' \Downarrow E, \Sigma''}{\texttt{if } B \texttt{ then } M_{true} \texttt{ else } M_{false}, \Sigma \Downarrow E, \Sigma''} \qquad \frac{B, \Sigma \Downarrow G, \Sigma'}{\texttt{if } B \texttt{ then } M_{true} \texttt{ else } M_{false}, \Sigma \Downarrow G, \Sigma'}$$

$$\frac{M, \Sigma \Downarrow m, \Sigma'}{x\texttt{:=}M, \Sigma \Downarrow \texttt{skip}, \Sigma'[x \mapsto m]} \qquad \frac{M, \Sigma \Downarrow G, \Sigma'}{x\texttt{:=}M, \Sigma \Downarrow G, \Sigma'} \qquad \frac{}{!x, \Sigma \Downarrow \Sigma(x), \Sigma}$$

$$\frac{M, \Sigma \otimes (x \mapsto 0) \Downarrow E, \Sigma' \otimes (x \mapsto n)}{\texttt{new } x.M, \Sigma \Downarrow E, \Sigma'} \qquad \frac{M, \Sigma \Downarrow \texttt{goto } x, \Sigma'}{\texttt{break } x.M, \Sigma \Downarrow \texttt{skip}, \Sigma'} \qquad \frac{M, \Sigma \Downarrow \texttt{skip}, \Sigma'}{\texttt{break } x.M, \Sigma \Downarrow \texttt{skip}, \Sigma'}$$

$$\frac{M, \Sigma \Downarrow \texttt{goto } x, \Sigma' \qquad \texttt{cont } x.M, \Sigma' \Downarrow E, \Sigma''}{\texttt{cont } x.M, \Sigma \Downarrow E, \Sigma''} \qquad \frac{M, \Sigma \Downarrow \texttt{skip}, \Sigma'}{\texttt{cont } x.M, \Sigma \Downarrow E, \Sigma'}$$

$$\frac{P, \mathcal{U} \otimes (f \mapsto F), \Sigma \Downarrow E, \Sigma'}{\texttt{let } f(x_1,\ldots,x_k) = F \texttt{ in } P, \mathcal{U}, \Sigma \Downarrow E, \Sigma'} \qquad \frac{\mathcal{U}(f) = F \qquad F[M_i/x_i], \mathcal{U}, \Sigma \Downarrow E, \Sigma'}{f(M_1,\ldots,M_k), \mathcal{U}, \Sigma \Downarrow E, \Sigma'}$$

Fig. 2. Operational semantics for IAL

$\mathbf{N} + \mathbf{B} + \mathbf{1}$. Let $\mathbf{G} = \{\texttt{goto } x \mid x \in \mathcal{L}\}$ be the set of non-local jumps. Let the set of *final forms* be $\mathbf{E} = \mathbf{V} + \mathbf{G}$. We assume $V \in \mathbf{V}, G \in \mathbf{G}$, etc. We also use an *environment* $\mathcal{U}$ which is a map from function-identifiers to terms. Let $\Sigma : \mathcal{C} \to \mathbb{Z}$ be a *store*, let $\Sigma \otimes (x \mapsto n)$ represent the extension of Σ to domain $\mathcal{C} + \{x\}$ such that $\Sigma \otimes (x \mapsto n)(x) = n$, and let $\Sigma[x \mapsto n]$ be a store equal to Σ except that $\Sigma[x \mapsto n](x) = n$. The operational semantics of the language are relations of the form $M, \mathcal{U}, \Sigma \Downarrow E, \Sigma'$, meaning term M in environment $\mathcal{U}$ and state Σ evaluates to final form $E \in \mathbf{E}$ and final state Σ'. If the environment is not used in the rule it will be omitted, for simplicity. The operational semantics is given in Fig. 2. Note that continue is expressive enough to encode iteration: $\texttt{while } M \texttt{ do } N \equiv \texttt{cont } y.\texttt{if } M \texttt{ then } N; \texttt{goto } y \texttt{ else skip}$. Also note that a notion of abnormal termination can be encoded with $\texttt{goto}$ $\texttt{abort}$, where $\texttt{abort}$ is a reserved label. With $\texttt{abort}$, assertions can be encoded as $\texttt{assert}(M) \equiv \texttt{if } M \texttt{ then skip else goto abort}$.

2.2 Game Semantics

In this section we will present a game-like model along the lines of [14], but with the important distinction that state will be modelled explicitly in a way rather similar to [15] and [16]. We can do this because of the greatly simplified role that locations can play in the language. The absence of var-type terms makes interference and bad variables impossible and supports a *global store* model.

A state $\Sigma : A \to \mathbb{Z}$ maps a set of names A to integer values. Given an alphabet $\mathcal{A}$ and a set of names A, a stateful sequence $s^{\Sigma\Sigma'}$ consists of a sequence $s \in \mathcal{A}^*$ and two states $\Sigma, \Sigma' : A \to \mathbb{Z}$. If $s = \epsilon$, the empty sequence, we require $\Sigma = \Sigma'$. We define the following operations on sets of stateful sequences, i.e. stateful languages: $S \cdot T = \{(s \cdot t)^{\Sigma\Sigma''} \mid s^{\Sigma\Sigma'} \in S, t^{\Sigma'\Sigma''} \in T\}$. Also,

$$S^{(0)} = \{\epsilon^{\Sigma,\Sigma} \mid \Sigma : A \to \mathbb{Z}\}, \quad S^{(k)} = S \cdot S^{(k-1)}, \quad S^* = \bigcup_{k \in \mathbb{N}} S^{(k)}.$$

If t, u are stateless sequences then we define $t \cdot s^{\Sigma\Sigma'} \cdot u = (t \cdot s \cdot u)^{\Sigma\Sigma'}$.

With every type U of the language we associate an alphabet $[\![U]\!]$:

$$[\![\texttt{int}]\!] = \{q\} \cup \mathbb{Z}, \quad [\![\texttt{bool}]\!] = \{q, t, f\}, \quad [\![\texttt{com}]\!] = \{q, a\}.$$

For function types we have

$$[\![T_1 \times \cdots \times T_n \to T]\!] = \sum_{i=1,n} [\![T_i]\!] + [\![T]\!]. \tag{1}$$

Terms $M : T$ are modelled by languages over alphabet

$$\mathcal{A}_M = [\![T]\!] + \sum_{\substack{U \ s.t. \\ \mathcal{V}(M) \cap \mathcal{F}_U \neq \emptyset}} [\![U]\!] + \sum_{y \in \mathcal{L}(M)} \texttt{go}^y. \tag{2}$$

To make the disjoint sum more explicit, we syntactically tag elements of $[\![U]\!]$ with the identifier x. The symbols in the alphabet are the so-called game-semantic "moves". They represent the *observable* actions that a term can perform. Every language that denotes a meaning of a term has a certain form, given by all its possible initial and final moves, called *bracketing moves*. If the final action belongs to the normal alphabet associated with the type, the trace is a complete computation leading to value a, and we denote it by $(\!|M|\!)_a$. Another possible final action is $\texttt{go}^x$ for some label x and it denotes an attempt to jump out of the scope of the term; we denote such traces $\{\!|M|\!\}$. The meaning of terms at ground type can be decomposed as:

$$[\![M : \texttt{com}]\!] = q \cdot (\!|M|\!)_a \cdot a + q \cdot \{\!|M|\!\}$$
$$[\![M : \texttt{bool}]\!] = q \cdot (\!|M|\!)_t \cdot t + q \cdot (\!|M|\!)_f \cdot f + q \cdot \{\!|M|\!\}$$
$$[\![M : \texttt{int}]\!] = \sum_{n \in \mathbb{Z}} q \cdot (\!|M|\!)_n \cdot n + q \cdot \{\!|M|\!\}.$$

$(\![\mathtt{skip}]\!) = \epsilon^{\Sigma\Sigma}, \quad \{\!|\mathtt{skip}|\!\} = \emptyset$

$(\![n]\!)_n = \epsilon^{\Sigma\Sigma}, \quad (\![m]\!)_n = \emptyset \text{ if } m \neq n, \quad \{\!|u|\!\} = \emptyset$

$(\![M_1; M_2]\!)_p = (\![M_1]\!) \cdot (\![M_2]\!)_p, \quad \{\!|M_1; M_2|\!\} = \{\!|M_1|\!\} + (\![M_1]\!) \cdot \{\!|M_2|\!\}$

$(\![M_1 \oplus M_2]\!)_p = \displaystyle\sum_{\substack{m,n,p\in\mathbb{Z} \\ m\oplus n=p}} (\![M_1]\!)_m \cdot (\![M_2]\!)_n, \quad \{\!|M_1 \oplus M_2|\!\} = \{\!|M_1|\!\} + \displaystyle\sum_{m\in\mathbb{Z}} (\![M_1]\!)_m \cdot \{\!|M_2|\!\}$

$(\![x\mathtt{:=}M]\!) = \displaystyle\sum_{n\in\mathbb{Z}} \left((\![M]\!)_n^{\Sigma\Sigma'}\right)^{\Sigma\Sigma'[x\mapsto n]}, \quad \{\!|x\mathtt{:=}M|\!\} = \{\!|M|\!\}$

$(\![!x]\!)_n = \epsilon^{\Sigma\Sigma} \text{ if } \Sigma(x) = n, \quad (\![!x]\!)_n = \emptyset \text{ if } \Sigma(x) \neq n, \quad \{\!|!x|\!\} = \emptyset$

$(\![\mathtt{new}\ x.M]\!) = \left((\![M]\!)^{\Sigma\otimes(x\mapsto 0),\Sigma'\otimes(x\mapsto n)}\right)^{\Sigma,\Sigma'}, \quad \{\!|\mathtt{new}\ x.M|\!\} = \left(\{\!|M|\!\}^{\Sigma\otimes(x\mapsto 0),\Sigma'\otimes(x\mapsto n)}\right)^{\Sigma,\Sigma'}$

$(\![\mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2]\!)_a = (\![M]\!)_t \cdot (\![M_1]\!)_a + (\![M]\!)_f \cdot (\![M_2]\!)_a,$

$\quad\quad \{\!|\mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2|\!\} = \{\!|M|\!\} + (\![M]\!)_t \cdot \{\!|M_1|\!\} + (\![M]\!)_f \cdot \{\!|M_2|\!\}$

$(\![\mathtt{goto}\ x]\!)_a = \emptyset, \quad \{\!|\mathtt{goto}\ x|\!\} = (\mathtt{go}^x)^{\Sigma\Sigma}$

$(\![\mathtt{break}\ x.M]\!)_a = (\![M]\!)_a + \{\!|M|\!\}_x, \quad \{\!|\mathtt{break}\ x.M|\!\} = \{\!|M|\!\}_y \cdot \mathtt{go}^y, x \neq y$

$(\![\mathtt{cont}\ x.M]\!)_a = \{\!|M|\!\}_x^* \cdot (\![M]\!)_a, \quad \{\!|\mathtt{cont}\ x.M|\!\} = \{\!|M|\!\}_x^* \cdot \{\!|M|\!\}_y \cdot \mathtt{go}^y, x \neq y.$

Fig. 3. Game-semantic evaluations

Intuitively, $(\![M]\!)_a, \{\!|M|\!\}$ are the observable effects of the actual computation that M carries out in order to produce a or jump, respectively.

For a term M we define a pattern-matching operator that extracts traces with a given initial and final states $(\![M]\!)^{\Sigma\Sigma'} \triangleq \{s \mid s^{\Sigma\Sigma'} \in (\![M]\!)\}$ and similarly for $\{\!|M|\!\}$. We also use the notation $\{\!|M|\!\} \triangleq \{\!|M|\!\}_x \cdot \mathtt{go}^x$ and we implicitly sum over all states Σ. Most of the semantic valuations are given in Fig. 3.

Note that constants have no observable side-effects and cannot jump. For break, normal termination is either the normal termination of M or a jump to the breaking label x; any other termination can only be a jump. For continue, any jump to x causes a restart of M, until it terminates normally or until it jumps to a different location than x.

As in [14] we only give a game-semantic definition for function application of a free function identifier, i.e. a function where the definition is not known. We choose not to present function application in general because it is too complex for this presentation, unrelated to the issue of predicate abstraction. In the absence of recursion β-redexes (i.e. function calls with known definitions) can be reduced operationally. It is fair to say that the entire apparatus of game semantics and the entire development to this point is necessary only insofar as it allows the formulation of this rule:

$$(\![f(M_1, \ldots, M_n)]\!)_k = q^f \cdot \left(\sum_{i=1}^n \sum_{a\in[\![T_i]\!]} q^{fi} \cdot (\![M_i]\!)_a \cdot a^{fi}\right)^* \cdot k^f$$

$$\{\!|f(M_1, \ldots, M_n)|\!\} = q^f \cdot \left(\sum_{i=1}^{n} \sum_{a \in [\![T_i]\!]} q^{fi} \cdot (\!|M_i|\!)_a \cdot a^{fi} \right)^* \cdot \left(\sum_{i=1}^{n} q^{fi} \cdot \{\!|M_i|\!\} \right).$$

Moves q^f, k^f are markers delineating the overall beginning and end of computation. Moves q^{fi}, a^{fi} are markers delineating the beginning and execution of each argument. A normal execution of a function is an arbitrary sequence of executions of its arguments. If one of the arguments causes a non-local jump then the function call terminates with that non-local jump. The locality of the jumps ensures that all jumps from M_is are either local or outside of the scope. It is not possible for arguments to cause jumps to each other.

Finally, for completeness, if $x : T$ is a base-type free variable then $(\!|x|\!)_a = x$ and $\{\!|x|\!\} = \emptyset$: its meaning is an unspecified action labelled with the variable.

We are mainly interested in proving safety properties. Suppose that there is a special label called `abort`. A term is abort-free if it has no occurrence of `goto abort`. We say that a term M is *safe* if for any abort-free context with a hole $\mathcal{C}[-]$ and for any state Σ we have $\mathcal{C}[M], \Sigma \Downarrow E, \Sigma', E \neq$ `goto abort`. The connection between the operational and game semantics is given by:

Theorem 1. *A term of* IAL *M is safe if and only if* $\{\!|M|\!\}_{\texttt{abort}} = \emptyset$.

The proof of this result is routine, similar to that in [4].

Example 1. Show `new x.f(c; x := !x + 2, assert (!x % 2 <> 0))` is safe.

This example illustrates the uniqueness of the game-semantic approach, because it requires reasoning about a non-trivial interaction between non-local function `f`, non-local procedure `c` and the store. The set of locations is $\mathcal{L} = \{x\}$ and the state is $\Sigma : \{x\} \to \mathbb{Z}$. For simplicity we denote the function $(x \mapsto n)$ simply as n. Following simple calculations we have

$$(\!|\texttt{c;x:=!x+2}|\!) = c \cdot \epsilon^{n,n+2} = c^{n,n+2} \quad (\!|\texttt{c;x:=!x+2}|\!) = \emptyset$$

$$(\!|\texttt{assert(!x\%2!=0)}|\!) = \epsilon^{2k,2k} \qquad \{\!|\texttt{assert(!x\%2!=0)}|\!\} = (\texttt{go}^{\texttt{abort}})^{2k+1,2k+1}.$$

Applying `f` gives:

$$(\!|\texttt{f(c;x:=!x+2,assert(!x\%2<>0))}|\!)$$

$$= q^f \cdot \left(\sum_n q^{f1} \cdot c^{n,n+2} \cdot a^{f1} + \sum_k q^{f2} \cdot \epsilon^{2k,2k} \cdot a^{f2} \right)^* \cdot a^f$$

$$\{\!|\texttt{f(c;x:=!x+2,assert(!x\%2<>0))}|\!\}$$

$$= q^f \cdot \left(\sum_n q^{f1} \cdot c^{n,n+2} \cdot a^{f1} + \sum_k q^{f2} \cdot \epsilon^{2k,2k} \cdot a^{f2} \right)^* \cdot \left(\sum_k q^{f2} \cdot (\texttt{go}^{\texttt{abort}})^{2k+1,2k+1} \right).$$

By a simple inductive argument, $\left(\sum_n q^{f1} \cdot c^{n,n+2} \cdot a^{f1} + \sum_k q^{f2} \cdot \epsilon^{2k,2k} \cdot a^{f2} \right)^*$ always produces traces of the form $s^{n,n+2k}$, therefore

$$(\!|\texttt{new x.f(c;x:=!x+2,assert(!x\%2<>0))}|\!) = q^f \cdot \left(q^{f1} \cdot c \cdot a^{f1} + q^{f2} \cdot a^{f2} \right)^* \cdot a^f$$

$$\{\!|\texttt{new x.f(c;x:=!x+2,assert(!x\%2<>0))}|\!\} = \emptyset,$$

since the rule for `new` forces the initial state to be 0 and removes the (only) location x from the state. According to Thm. 1 this means the term is safe. Note that if we take the set of integers to be finite the set of state annotations is also finite and the formula can be mechanically verified.

3   Predicate Abstraction

The key problem of automatic software verification is that the set of all possible states Σ is very large. If we restrict IAL to finite k-bit integers, then a set of states over n variables has, obviously, 2^{nk} elements. Predicate abstraction in game semantics is about reducing the size of this set, in a way that is compatible with the compositional (denotational) structure of the semantics and which can still model the subtle interplay between store and procedural behaviour. To further simplify the presentation we will only consider predicate abstraction for assignments that only use *pure expressions* on the RHS, i.e. expressions that do not change the state while returning a value. This is not a substantial restriction, as all programs can be converted to that form using assignment to intermediate values.

We abstract a state Σ in the standard way (e.g. [9]) by representing it as a set of predicates over $\mathrm{dom}(\Sigma)$. If σ is approximated by p predicates then the number of possible values is 2^p, which can be far smaller than 2^{nk}. We denote an abstracted state by Ψ.

We introduce the following notations. Given a set of states $\mathcal{S} = \{\Sigma \mid \Sigma : \mathcal{L} \to \mathbb{Z}\}$ over locations $\mathcal{L}$, let $\mathbb{P}_{\mathcal{L}}$ be the set of all predicates definable using its locations as variables, and let $\mathcal{P}_{\mathcal{L}} \in \mathbb{P}_{\mathcal{L}}^*$ a (finite) list of its elements, constituting the predicate abstraction of $\mathcal{S}$. The predicates $\Psi \in \mathcal{P}_{\mathcal{L}}$ are called *abstract states*. A set of abstract states is *satisfiable* written $\mathrm{sat}(\Psi_0, \ldots, \Psi_k)$ if there is an assignment of their variables that makes each Ψ_i true; we call such predicates that are simultaneously satisfiable *compatible*.

We define pa-traces similar to stateful traces, $s^{\Psi\Psi'}$, with concatenation of pa-languages defined as $S \cdot T = \{(st)^{\Psi\Psi'} \mid s^{\Psi\Psi_0} \in S, t^{\Psi'_0\Psi'} \in T, \mathrm{sat}(\Psi_0, \Psi'_0)\}$. Note that concatenation of pa-traces is non-deterministic, due to the possible choices for Ψ_0, Ψ'_0, unlike stateful trace concatenation which is deterministic. Exponentiation and iterated closure are defined similarly to stateful traces.

Let $\mathcal{E}_N$ and $\mathcal{E}_B$ be the languages of integer and boolean expressions constructed from constants, arithmetic and logic operators and uninterpreted variables. The predicate-abstracted semantics is defined in terms of pa-traces and is structurally similar to that of the original game semantics. $\overline{[\![\mathtt{int}]\!]} = \{q\} \cup \mathcal{E}_N$, $\overline{[\![\mathtt{bool}]\!]} = \{q\} \cup \mathcal{E}_B$, $\overline{[\![\mathtt{com}]\!]} = \{q, a\}$. Function-type and term alphabets are analogously to Eqns. 1 and 2. Note that the sub-alphabet of result moves is expanded from the set of all *values* of a given type to the set of all *syntactic expressions* over $\mathcal{L}$ of a given type. Trace decompositions are analogous to game semantics: $\overline{[\![M : \mathtt{com}]\!]} = q \cdot \overline{(\![M]\!)}_a \cdot a + q \cdot \overline{\{\![M]\!\}}$, and so on for the other types. The semantic rules for constants, sequential composition, control and function application are also analogous to those of the original game semantics. We only

present the rules that are substantially different: branching, arithmetic and logic, assignment and dereferencing, local variable.

We introduce the notation $\overline{(\![M]\!)}_B^{\langle \Psi_0 \Psi_1 \rangle} \triangleq \left(\overline{(\![M]\!)}_B^{\Psi_0 \Psi_1} \right)^{\Psi_0 \Psi_1}$ to identify particular traces. Note that $\overline{(\!-\!)}^{\Psi_0, \Psi_1}$ is a trace-selection operator whereas $(-)^{\Psi_0, \Psi_1}$ is an annotation. The pa-semantics of branching is:

$$\overline{(\![\texttt{if } M \texttt{ then } M_1 \texttt{ else } M_2]\!)}_a$$

$$= \sum_{\substack{B \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1, B)}} \overline{(\![M]\!)}_B^{\langle \Psi_0 \Psi_1 \rangle} \cdot \overline{(\![M_1]\!)}_a + \sum_{\substack{B' \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1', \neg B')}} \overline{(\![M]\!)}_{B'}^{\langle \Psi_0' \Psi_1' \rangle} \cdot \overline{(\![M_2]\!)}_a$$

$$\overline{\{\![\texttt{if } M \texttt{ then } M_1 \texttt{ else } M_2]\!\}}$$

$$= \overline{\{\![M]\!\}} + \sum_{\substack{B \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1, B)}} \overline{(\![M]\!)}_B^{\langle \Psi_0 \Psi_1 \rangle} \cdot \overline{\{\![M_1]\!\}} + \sum_{\substack{B' \in \mathcal{E}_B \\ \mathrm{sat}(\Psi_1', \neg B')}} \overline{(\![M]\!)}_{B'}^{\langle \Psi_0' \Psi_1' \rangle} \cdot \overline{\{\![M_2]\!\}},$$

Note that the guard M evaluates to a *syntactic* expression B, rather than a value. The branch to be executed is chosen depending on whether the expression is compatible with the state or whether its negation is. Note that it is possible that both conditions are satisfied, case in which the branching becomes non-deterministic. Arithmetic and logic operators evaluate to a syntactic expression rather than a value:

$$\overline{(\![M_1 \oplus M_2]\!)}_{E_1 \oplus E_2} = \overline{(\![M_1]\!)}_{E_1} \cdot \overline{(\![M_2]\!)}_{E_2},$$

$$\overline{\{\![M_1 \oplus M_2]\!\}} = \overline{\{\![M_1]\!\}} + \overline{(\![M_1]\!)} \cdot \overline{\{\![M_2]\!\}}.$$

The rule for assignment is:

$$\overline{(\![x\texttt{:=}M]\!)} = \sum_{E \in \mathcal{E}_N} \left(\overline{(\![M]\!)}_E^{\Psi \Psi'} \right)^{\Psi \Psi''}, \quad \overline{\{\![x\texttt{:=}M]\!\}} = \overline{\{\![M]\!\}},$$

where $\mathrm{sat}(\Psi'', \Psi'[E/x])$.

Note that after assignment a new pa-state Ψ'' must be chosen, which is compatible to the old state in which x has become E. Note that the choice of Ψ'' can introduce non-determinism in the interpretation. The pa-semantics of assignment is non-deterministic, unlike the game-semantic interpretation.

Dereferencing returns x, seen as a syntactic expression:

$$\overline{(\![!x]\!)}_x = \epsilon^{\Psi \Psi}, \quad \overline{(\![!x]\!)}_E = \emptyset \text{ if } E \neq x, \quad \overline{\{\![!x]\!\}} = \emptyset.$$

Local variable introduction must cope with the fact that the bound variable cannot appear outside of its context, therefore the pa-states inside the block must come from a different set than that used outside the block, which uses a smaller set of locations.

$$\overline{(\![\texttt{new } x.M]\!)} = \left(\overline{(\![M]\!)}^{\Psi_0 \Psi_1} \right)^{\Psi_0', \Psi_1'},$$

$$\overline{\{\![\texttt{new } x.M]\!\}} = \left(\overline{\{\![M]\!\}}^{\Psi_0 \Psi_1} \right)^{\Psi_0', \Psi_1'}.$$

where $\Psi_0', \Psi_1' \in \mathcal{P}_{\mathcal{L}}$, $\Psi_0, \Psi_1 \in \mathcal{P}_{\mathcal{L}+\{x\}}$, $\mathrm{sat}(\Psi_0, \Psi_0', x = 0), \mathrm{sat}(\Psi_1, \Psi_1')$. As in the case of assignment, the choice of updated state is not necessarily deterministic. It is assumed that local variables are initialized to zero.

Example 2. We reconsider the Example 1 program, using pa-semantics to show that `new x.f(c; x := !x + 2, assert (!x % 2 <> 0))` is safe.

Assume the singleton predicate set $\mathcal{P}_{\{x\}} = \{\mathrm{even}(x)\}$, so the only possible abstracted states are $e = \mathrm{even}(x)$ and $o = \neg\mathrm{even}(x)$. Following simple calculations we have

$$\overline{(\!|\,!\mathtt{x+2}|\!)}_{x+2} = \epsilon^{o,o} + \epsilon^{e,e}, \quad \overline{\{\!|\,!\mathtt{x+2}|\!\}}_{x+2} = \emptyset, \quad \overline{(\!|\,!\mathtt{x+2}|\!)}^{o,o}_{x+2} = \epsilon, \quad \overline{(\!|\,!\mathtt{x+2}|\!)}^{e,e}_{x+2} = \epsilon$$

$$\overline{(\!|\mathtt{x:=!x+2}|\!)} = \left(\overline{(\!|\,!\mathtt{x+2}|\!)}^{o,o}_{x+2}\right)^{o,\Psi} + \left(\overline{(\!|\,!\mathtt{x+2}|\!)}^{e,e}_{x+2}\right)^{e,\Psi'},$$

where $\mathrm{sat}(\neg\mathrm{even}(x+2), \Psi)$, $\mathrm{sat}(\mathrm{even}(x+2), \Psi')$, so $\Psi = e, \Psi' = o$. Therefore $\overline{(\!|\mathtt{x:=!x+2}|\!)} = \epsilon^{o,o} + \epsilon^{e,e}$. Also, $\overline{\{\!|\mathtt{x:=!x+2}|\!\}} = \emptyset$. The two arguments are

$$\overline{[\![\mathtt{c;x:=!x+2}]\!]} = q \cdot \overline{(\!|\mathtt{c;x:=!x+2}|\!)} \cdot a + q \cdot \overline{\{\!|\mathtt{c;x:=!x+2}|\!\}} = qca^{o,o} + qca^{e,e}$$

$$\overline{[\![\mathtt{assert(!x\%2<>0)}]\!]} = q \cdot a^{e,e} + (q \cdot \mathsf{go}^{\mathsf{abort}})^{o,o}.$$

Applying f gives:

$$\overline{[\![\mathtt{f(c;x:=!x+2,assert(!x\%2<>0))}]\!]}$$
$$= q^f \cdot \left(q^{f1} \cdot c^{e,e} \cdot a^{f1} + q^{f1} \cdot c^{o,o} \cdot a^{f1} + q^{f2} \cdot \epsilon^{e,e} \cdot a^{f2}\right)^* \cdot a^f$$

$$\overline{\{\!|\mathtt{f(c;x:=!x+2,assert(!x\%2<>0))}|\!\}}$$
$$= q^f \cdot \left(q^{f1} \cdot c^{e,e} \cdot a^{f1} + q^{f1} \cdot c^{o,o} \cdot a^{f1} + q^{f2} \cdot \epsilon^{e,e} \cdot a^{f2}\right)^* \cdot \left(q^{f2} \cdot (\mathsf{go}^{\mathsf{abort}})^{o,o}\right).$$

Obviously the iteration $\left(q^{f1} \cdot c^{e,e} \cdot a^{f1} + q^{f1} \cdot c^{o,o} \cdot a^{f1} + q^{f2} \cdot \epsilon^{e,e} \cdot a^{f2}\right)^*$ always produces either traces of the form $s^{e,e}$ or $s^{o,o}$, therefore

$$\overline{[\![\mathtt{new\ x.f(c;x:=!x+2,assert(!x\%2<>0))}]\!]} = q^f \cdot \left(q^{f1} \cdot c \cdot a^{f1} + q^{f2} \cdot a^{f2}\right)^* \cdot a^f$$
$$\overline{\{\!|\mathtt{new\ x.f(c;x:=!x+2,assert(!x\%2<>0))}|\!\}} = \emptyset,$$

since the rule for `new` forces the initial state to be compatible with $x = 0$ (i.e. $\Psi = e$); the outer set of predicates is defined over the empty set of locations and is omitted. Note that in this (not typical) case the pa-semantics and the game semantics give the same interpretation.

3.1   Formal Properties

The technical hurdle is formulating the pa-semantics; with the definitions in place proving the relevant technical properties is a routine exercise.

This ancillary result is important towards proving decidability.

Proposition 1. *For any $M : T$, $T \in \{\mathtt{bool}, \mathtt{int}\}$ there is only a finite set of syntactic expressions E such that $\overline{(\!|M : T|\!)}_E \neq \emptyset$.*

Proposition 2 (Decidability). *If $\mathcal{P}_\mathcal{L}$ is finite then $\overline{\{\![M]\!\}} = \emptyset$ is decidable.*

The proof relies on the fact that, given a finite set of pa-state annotations Ψ, and considering the finitary encoding of the alphabet (from Prop. 1) the pa-semantics accepts a regular-language formulation. Let us write $\overline{\{\![M]\!\}}_{\langle \mathcal{P} \rangle}$ when we need to emphasise that a pa-semantics is over predicate set $\mathcal{P}$.

Proposition 3 (Monot.). *If $\mathcal{P} \subseteq \mathcal{P}'$ and $\overline{\{\![M]\!\}}_{\langle \mathcal{P} \rangle} = \emptyset$ then $\overline{\{\![M]\!\}}_{\langle \mathcal{P}' \rangle} = \emptyset$*

The contra-positive has an immediate proof; if $\overline{\{\![M]\!\}}_{\langle \mathcal{P}' \rangle}$ has a trace then removing a predicate does not invalidate any of the satisfiablity conditions, therefore $\overline{\{\![M]\!\}}_{\langle \mathcal{P} \rangle}$ will have a trace. The monotonicity property states that "improving" the abstraction does not remove any possible failure traces.

Proposition 4 (Correctness). *If $\mathcal{P} = \{x = n \mid x \in \mathcal{L}, n \in \mathbb{Z}\}$ then $\overline{\{\![M]\!\}} = \emptyset$ iff $\{\![M]\!\} = \emptyset$.*

The proof is immediate, as the Ψ's are a precise predicate representations of the state Σ in the stateful formulation of the game semantic model.

From Correctness and Monotonicity, along with Thm. 1 it follows that

Theorem 2 (Soundness). *If $\overline{\{\![M]\!\}} = \emptyset$ then M is safe.*

Thm. 2 and Prop. 2 state that any finite PA semantics is a sound and effective approximation for the concrete semantics, and it can be used for automatic proving of safety properties of IAL terms.

4   Heuristics

Our experimental model checker MAGE implements automatic verification algorithms within the framework described in previous sections. We use a finite set of predicates of form $\Psi = \bigwedge_{P \in \mathcal{P}} \delta_i(P)$ where each δ_i is either the identity or the negation operation and each P is a proposition over the set of locations. Such predicates can be efficiently represented as bit-vectors.

4.1   Internal and External Compositionality

Games-based verification tools are compositional in the sense that they can handle open terms (see earlier examples); call this kind of compositionality *external*. Additionally, games-based tools are *internally compositional*, i.e. the model of a term is built inductively from the models of its sub-terms. This approach is helpful because it allows the modification of the abstraction scheme within the term being constructed; some branches of the syntax tree can be heavily abstracted while others can be much more precise. This feature has been discussed in the context of games-based data-abstraction and refinement [4,6], and it can be used again to great effect with predicate abstraction by changing the predicate set within the term.

4.2   Flexibility and Efficiency

The pa-semantics allows the predicate set to change at every composition point in the program — i.e. between every sub-term and its successor in the parse tree. This flexibility can be exploited by removing from the pa-state Ψ predicates P deemed irrelevant and reintroducing them whenever precision needs to be improved. It is well known that minimizing the predicate set size is essential to avoid a state explosion; moreover, n predicate-bits are typically much more expensive to maintain than an n-bit state in conventional model checking, because each bit represents an arbitrarily complex predicate.

Predicate annotations. Running a satisfiability check over a current state and all possible next-states, and allowing the entire predicate set to change at every composition is too expensive in general. To make our treatment of variable abstraction schemes more perspicuous, we shall use syntactic annotations "`newp`" and "`endp`" to delimit predicate scopes at the level of the source code. This is important because for any assignment we can track the state-change precisely by only using two predicates (i.e. a two-bit vector), representing the state before and after the assignment. For example, in the code fragment below no more than two predicates are needed at any one time to track the following execution accurately and validate the assertion (`c:com` is a free procedure identifier).

```
newp (x = 0); x := 0; c;
newp (x = 1); x := !x + 1; c; endp (x = 0);
newp (x = 11); x := !x + 10; c; endp (x = 1);
newp (x = 111); x := !x + 100; c; endp (x = 11);
assert(x = 111); endp (x = 111)
```

Another simplifying restriction we impose is that predicate scopes are well nested and use instead "`letp` p `in` M" annotations. Next-state calculation (which predicates can change valuation at a given program point), and identification of relevant predicates for current-state tests become much simpler because the predicates form a stack that can be mapped into the standard program stack. On the other hand, a disadvantage of nested scoping is that a series of overlapping scopes cannot always be kept tight. In the same example the maximum size of the bit vector used to represent the pa-state is now four:

```
letp (x = 0) in x := 0; c;
letp (x = 1) in x := !x + 1; c;
letp (x = 11) in x := !x + 10; c;
letp (x = 111) in x := !x + 100; c;
assert(x = 111)
```

4.3   Predicate Scope

Our experimental tool, MAGE resolves satisfiability tests with the external SMT engine YICES[2]. Two fully automatic predicate annotation schemes are used, both in the compositional "`letp`" style:

---

[2] `http://yices.csl.sri.com/`

1. All (pure) conditional expressions in the program are predicates in the model, each given maximal scope to maximize the chance of a successful check (but expensive in situations where a scope contains many conditionals).
2. Conditionals are made predicates with minimal scope initially; checking is therefore much faster and much more likely to be inconclusive. A refinement loop is added to expand the scope of some predicates (see Section 4.4).

It is this issue of *predicate scope* that the internal compositionality of the game-based formulation exposes and allows to be manipulated to the advantage of the verification process: for success the selected predicates must be both adequate and have sufficient scope; but too many predicates with excessive scope will make checking infeasible.

Example 3 (Number magic). The magician asks the stooge to think of an n, double it, add 50, divide by 2, subtract n and add 100. The `distract:com` procedure is just that, an irrelevant diversion.

```
new m.new n.n := stooge(); distract;
m := !n + !n;  distract; assert(m = 2 * n);
m := !m + 50;  distract; assert(m = 50 + 2 * n);
m := !m / 2;   distract; assert(m = 25 + n);
m := !m - !n;  distract; assert(m = 25);
m := !m + 100; distract; assert(m = 125)
```

The checks on magic answer m provide the intermediate invariants needed to prove the trick by predicate abstraction and the distractions prevent MAGE treating the assignment sequence as a single basic block (which makes the proof trivial)!

This is easily verified by MAGE using maximally scoped predicates. But notice that with five conditionals and hence five predicates (and no clever optimizations) the satisfiability of 2^5 possible next-states must be tested at each composition point in the pa-model where the state can change. This takes around 250 seconds.

MAGE can also verify the term much faster, in about 2 seconds, using minimally scoped predicates, widened just enough to include the assignment that makes it true and the assertion that declares it. This is achieved by preserving the valuations of predicates from the end of their `letp` scope until the next assignment. For example, in `letp (m = 25) in (m := !m - !n; distract();` `assert(m = 25));` at the assignment-composition, the valuation `m = 25 + n` from the previous scope is compatible only with next-state `m = 25`, as per the definition of pa-trace concatenation.

Hence the problem of having to nest overlapping scopes is eliminated as the model of `letp` p `in` M_1; `letp` p `in` M_2 can reach exactly the same states as that of `letp` p `in` $(M_1;M_2)$ because pa-trace concatenation always kicks in at the boundary between `letp`s.

There is considerable potential for further speed up the implementation by developing tighter solver integration; the incremental-SMT approach to predicate

abstraction is fully compatible with our approach and would shrink run-times
by a substantial factor although it cannot in itself avoid the state explosions.

4.4   Counter-Example Guided Scope Refinement

Counterexample certification checks trace *feasibility* of a pa-trace. Pa-trace con-
catenation offers a *local* check of compatibility between pa-states, but each such
concatenation point is a source of non-determinism. After several such concate-
nations the global trace from start to end may be actually not possible.

Consider for example the following code, with abstracting predicates written
explicitly as program annotation in `letp` style:

```
letp (x > y) in letp (x < y) in letp (z <> 2) in
     x := n; y := m;
     if !x > !y then z := 1 else z := 0;
     if !x < !y then z := !z + 1 else skip;
     assert(!z <> 2)
```

The second assignment to `z` may go from a pa-state in which `z <> 2` to a pa-
state in which `z <> 2` or to one in which `z = 2`. Because these distinctions
cannot be made locally, the feasibility check must involve a *global* satisfiability
test of the entire concatenation. In this case, we must check whether all predi-
cates `x1 = n`, `y1 = m`, `x1 <= y1`, `z1 = 0`, `x1 < y1`, `z2 = z1 + 1`, `z2 = 2`
are compatible.

Refinement is realised by modifying the predicate abstraction with the aim
of eliminating some infeasible counterexamples and the guarantee of not intro-
ducing new infeasible counterexamples. In our framework it can be achieved
by adding predicates or extending the scope of existing predicates. MAGE in
predicate-scope refinement mode begins by using the guard of each if statement
(implicit in assert) as a tightly-scoped predicate, represented by the annotation:

$$\text{if } B \text{ then } M \text{ else } N \Rightarrow \text{if } (\texttt{letp } B \text{ in } B) \text{ then } M \text{ else } N$$

Note that the unannotated term and the tight annotation have identical models
as the predicates never live long enough to be absorbed into the global state.
So for the simple algorithm in MAGE (with no interpolation or other methods
for adding predicates that are not conditionals), refinement now simply means
extending the scope of some `letp` that appears in an infeasible counterexample!

Example 4. Consider the safe term

```
new x.new y.new z.
     x := !i; y := !i;
     if letp (x > y) in !x > !y
     then z := !x - !y; assert(letp (z > 0) in !z > 0)
     else skip
```

Checking generates a counterexample that needs to satisfy `x1 = i1`, `y1 = i2`,
`x1 > y1`, `z1 = x1 - y1`, `z1 > 0` which is infeasible. Maximizing the two pred-
icate scopes would eliminate the counterexample in one step, but by expanding
them more gradually we arrive at the following provably safe scopes:

```
new x.new y.new z.x := !i; y := !i;
     letp (x > y) in if !x > !y
     then letp (z > 0) in z := !x - !y; assert(!z > 0)
     else skip
```

The tight scopes require a running time of 0.75 sec. while the loose scope requires 7.5 sec. of execution.

Note that this scheme is guaranteed to terminate and the program will be verified if it is verifiable when annotated with maximally-scoped conditionals. The gradual scope expansion creates two problems: more refinement iterations and the need for heuristics regarding which letp to expand. By a depth-first expansion scheme the tightest safe annotation will be found. This makes ours an alternative approach to the idea of *predicate minimization* used in other tools [17] so the burden of the expanded predicates on verification in other parts of the model should be minimized. The problem of more iterations is mitigated if model checking on the intermediate refinements is restricted to testing for the presence of the infeasible counterexample.

5 Further Work

We believe we are only beginning to exploit the new possibilities of attacking state-space explosion through internal compositionality. The next instance of the tool will incorporate at least a form of refinement by delayed SMT: we will extend the semantics and tool to allow the precision of a fixed pa to be gradually improved by delaying SMT tests for up to a specified number of concatenations. This achieves the same effect as expanding the scope of each predicate without actually increasing the state, but at the cost of more false counterexamples.

There is now a real potential to combine game-semantic models with more complex state-representation assertion languages, such as those arising from separation logic [18] and target heap-oriented programs compositionally. We are currently examining this, as well as game-semantic models of more realistic programming languages.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside Microsoft. In: IFM, pp. 1–20 (2004)
2. Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 25–26. Springer, Heidelberg (2005)
3. Abramsky, S., Ghica, D.R., Murawski, A.S., Ong, C.-H.L.: Applying game semantics to compositional software modeling and verification. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 421–435. Springer, Heidelberg (2004)
4. Dimovski, A., Ghica, D.R., Lazić, R.S.: Data-abstraction refinement: A game semantic approach. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 102–117. Springer, Heidelberg (2005)

5. Ghica, D.R., Murawski, A.S.: Compositional model extraction for higher-order concurrent programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 303–317. Springer, Heidelberg (2006)
6. Bakewell, A., Ghica, D.R.: On-the-fly techniques for game-based software model checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 78–92. Springer, Heidelberg (2008)
7. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)
8. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163(2), 285–408 (2000)
9. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
10. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. Electr. Notes Theor. Comput. Sci. 3 (1996)
11. Laird, J.: A fully abstract game semantics of local exceptions. In: LICS, pp. 105–114 (2001)
12. Reynolds, J.: The craft of programming. Prentice-Hall, Englewood Cliffs (1981)
13. Pitts, A.M.: Reasoning about local variables with operationally-based logical relations. In: O'Hearn, P.W., Tennent, R.D. (eds.) Algol-Like Languages, July 1996, vol. 2, pp. 173–193. Birkhauser, Basel (1997); reprinted from Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science, Brunswick, NJ, July 1996, pp. 152–163 (2006)
14. Ghica, D.R., McCusker, G.: The regular-language semantics of second-order Idealized Algol. Theor. Comput. Sci. 309(1-3), 469–502 (2003)
15. Ong, C.H.L.: Observational equivalence of 3rd-order Idealized Algol is decidable. In: LICS, pp. 245–256 (2002)
16. Laird, J.: A game semantics of names and pointers. Annals of Pure and Applied Logic 151(2-3), 151–169 (2008); first Games for Logic and Programming Languages Workshop
17. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 19–34. Springer, Heidelberg (2003)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)

Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications[*]

Hillel Kugler[1] and Itai Segall[2,**]

[1] Computational Biology Group, Microsoft Research, Cambridge, UK
hkugler@microsoft.com
[2] Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
itai.segall@weizmann.ac.il

Abstract. Synthesis is the process of automatically generating a correct running system from its specification. In this paper, we suggest a translation of a Live Sequence Chart specification into a two-player game for the purpose of synthesis. We use this representation for synthesizing a reactive system, and introduce a novel algorithm for composing two such systems for two subsets of a specification. Even though this algorithm may fail to compose the systems, or to prove the joint specification to be inconsistent, we present some promising results for which the composition algorithm does succeed and saves significant running time. We also discuss options for extending the algorithm into a sound and complete one.

1 Introduction

Automatic synthesis of systems directly from their specification has been a dream for many researchers. In the dream, a specifier is faced with an expressive yet intuitive specification language, in which she specifies the requirements from her system. All the rest will then happen automatically – by clicking a button, the specification will automatically be checked for consistency, and if found consistent, a system that is correct-by-construction will be generated, i.e., a system that is guaranteed to satisfy the specification. On the way towards realizing this dream, one must first choose a specification language that is both expressive and intuitive, and then build strong and fast algorithms for synthesizing systems from this language. Synthesis raises major challenges in terms of the inherent complexity of the problem and the required methodological development approach. Compositional synthesis, in which two synthesized systems may easily be composed into one large system, may help in addressing these challenges. By synthesizing small parts of the specification separately, and composing the intermediate results, one may save significant running time. Moreover, specifications

---

are usually constructed by a team and evolve over time, by introducing more requirements and modifying existing ones. A compositional approach to synthesis supports such evolution by reusing results for existing parts of the specification, rather than having to synthesize a complete system whenever the specification is modified. Another common way of tackling the high complexity of synthesis is by introducing semi-automatic algorithms. Such algorithms require some interaction with the user, but may perform much better by exploiting the user's understanding of the system. In a compositional synthesis algorithm this can be done by leaving the choice of the parts that should be separately synthesized to the user.

Live Sequence Charts (LSCs) [8] have been introduced as a highly expressive extension of Message Sequence Charts [18]. LSCs are multi-modal charts that distinguish between behaviors that may happen (existential, cold) and those that must happen (universal, hot). LSCs are highly expressive, and different translations of LSCs into temporal logic have been suggested (see, e.g., [9,21]). On the other side, being visual in nature, we believe the language is highly intuitive. Thus, LSCs were suggested as an expressive and intuitive specification language to use for synthesis in [13]. Despite research efforts on synthesizing systems from LSCs, e.g., [15,5], practical application to real-world systems has not yet been achieved.

In this paper, we propose a representation of LSC specifications as two-player game structures, in which a winning strategy for the system is equivalent to a reactive system satisfying the requirements. This representation is then synthesized into a reactive system using an approach similar to that proposed in [28,29]. We further propose a method for composing two synthesized systems. This method consists of an algorithm that is sound but not complete, and an algorithm that is complete but not sound. Therefore, it might fail to compose systems, or to prove their specifications to be inconsistent. However, we do provide several test cases for which it does succeed in creating a system for the entire specification, or prove the entire specification to be inconsistent, in running time significantly faster than that of non-compositional synthesis. We also briefly describe an extension of the approach that is sound and complete. This extension may be problematic in terms of running time and implementation, therefore it is given in this paper mainly for completeness of the approach, rather than as a full replacement for synthesis of composite specifications.

This work focuses on a subset of LSCs that includes only messages, and assumes that main charts include only messages controlled by the system. We also assume that no LSC has multiple copies simultaneously open during runtime. Finally, all messages in the specification are assumed to be synchronous, i.e., the event of sending a message and receiving it are simultaneous.

We implemented the approach introduced here as part of the new Scenario-Based Tool [33] developed at Microsoft Research Cambridge, using TLV [31] for the symbolic computations.

Some details of implementation, proofs, and notations are omitted from this version of the paper due to lack of space. See [23] for more details.
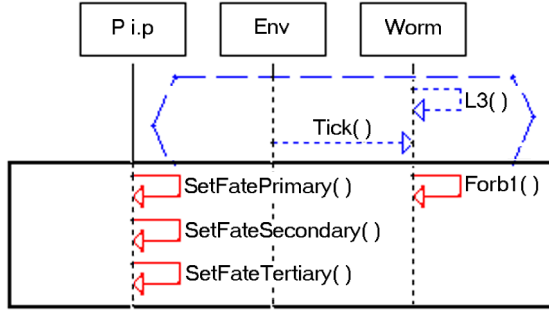
Fig. 1. An example LSC

2   Preliminaries

2.1   Live Sequence Charts

Live sequence charts (LSCs) [8] are an extension of message sequence charts (MSCs) [18]. LSCs, like MSCs, contain vertical lines, termed *lifelines*, which denote objects, and *events*, involving one or more lifelines. The most basic construct of the language are *messages*: a message is denoted by an arrow between two lifelines (or from a lifeline to itself), representing the event of the source object sending a message to the target object. A typical LSC consists of a prechart (denoted by a blue dashed hexagon), and a main chart (denoted by a solid frame). The intended semantics is that whenever the prechart is satisfied in a run of the system, eventually the main chart must also be satisfied. The synthesis method presented here focuses on messages, and does not currently support any of the more advanced constructs of the language, such as conditions, loops, etc.

Any object taking part in the specification is either controlled by the system, or by the environment. A message is said to be a *system (environment) message*, i.e., controlled by the system (environment), if it is sent from an object controlled by the system (environment).

An example of an LSC appears in Fig. 1. The LSC refers to three objects, `Env` representing the environment, and `Worm` and `Pi.p` representing two system objects. The LSC states that whenever `Worm` sends the message `L3` to itself, and then `Env` sends `Tick` to `Worm` (in this order), then `Pi.p` should send itself the messages `SetFatePrimary`, `SetFateSecondary` and `SetFateTertiary` (in this order), and `Worm` should send itself the message `Forb1`. Note that in the main chart there is no explicit order between the `Pi.p` messages and that of the `Worm`. Also note that if one of the main chart messages occurs before the prechart ends (e.g., after `L3` is sent, but before `Tick`), then the prechart is cold-violated and gracefully closed. This is considered legal behavior. If, however, the prechart completes, then the main chart messages must be sent in the correct order. Otherwise, this is considered a violation of the specification.

An operational semantics and an execution technique termed *play-out* was defined for the LSC language in [16]. Play-out remembers at each point in time

for each LSC the current *cut* (intuitively, a marker of what has already happened, and what not). It also maintains the set of *active* LSCs (those for which the prechart has been satisfied, but not the main chart). The set of all LSCs and their cuts is termed a *configuration*. At each step, the play-out mechanism chooses one message that is *enabled* in some active LSC (i.e., appears right after the current cut), and not violating in any others (a message is violating if it appears in an active chart but is not enabled), and executes it. Stronger mechanisms, termed *smart play-out* and *planned play-out* are introduced in [14,17]. These are initiated following each environment step, and look for a sequence of system events to perform in response (termed *superstep*), in order to drive the system to a stable state (one in which no LSC is active). However, looking only one superstep, or a finite number of supersteps, ahead, is not sufficient either. An example for this is given in [12]. This leads to the synthesis problem, i.e., given an LSC specification, finding a reactive system that adheres to the specification, or proving one does not exist.

2.2 Game Structures and Strategies

We view the synthesis problem as a two-player game between the system and environment, as formulated in a game structure. We modify the game structure and strategy definitions from [29] to reflect games in which the system is the first player. Intuitively, a game structure is a tuple $G : \langle V, X, Y, \Theta, \rho_s, \rho_e, \varphi \rangle$, where V represents the set of *state variables*, X is the set of system-controlled variables, and Y is the set of environment-controlled variables. Θ is the initial condition. ρ_s and ρ_e represent the transition relations of the system and environment, resp. The transition relation of the system depends only on the current state, whereas that of the environment may depend also on the system's transition. Finally, φ is the winning condition, of the form $\varphi = \Box\Diamond q$, where q is a state formula.

A *strategy* is a partial function mapping a series of states to a set of possible system actions. A run is *compliant* with a strategy if each step taken by the system is one allowed by the strategy. A strategy is *winning for the system* if any run, in which the environment takes only legal steps (i.e., ones allowed by the game definition), and is compliant with the strategy, is winning for the system, i.e., satisfies φ. Finally, a game structure is *realizable* if there exists a strategy that is winning for the system from any initial state (one satisfying Θ).

3   The Synthesis Problem

The synthesis problem is defined as follows. Given an LSC specification, determine whether there exists a reactive system that satisfies the specification, and generate one if so. Such a system will be called a *synthesized system*. This synthesized system must fulfill two requirements: infinitely often it must listen for environment events, and, it must never violate the specification. Note that violation here refers both to explicit violations of the requirements (safety), and to cases in which a step that must happen never does (liveness).

Two major distinct views can be taken when considering synthesis from LSCs. According to the first view, inspired by [16], the synthesized system is a direct execution engine for the specification, that never violates it. According to the second view, adopted in this work, the system need not execute the specification directly – it may take any action, as long as the two requirements above hold. We will refer to the problem addressed in this paper as *synthesis*, and to the other interpretation as *non-violating execution*.

In practice, the difference between the two translates to the choice of steps from a given state. In synthesis, the system may perform any step it deems necessary, while in non-violating execution, a message may be sent only if it is enabled in some active main chart, as defined in the operational semantics [16].

Our interpretation of the synthesis problem treats the specification as more under-specified – the specifier states things that may and must happen in the system, but anything unconstrained may also happen. In non-violating execution, an event may happen only if explicitly specified so. Note that in non-violating execution, a specification may be unrealizable, but become realizable by adding another LSC (or set of LSCs). In synthesis, however, specifications are monotonic. If a specification is unrealizable, then so is every extension thereof, and vice versa, a synthesized system for a specification may also serve as a synthesized system for any subset of it. This monotonicity gives rise to the issue of composition – given synthesized systems for two specifications, find a system for the unified specification.

Non-violating execution may seem more appropriate for finalized specifications. However, for intermediate stages the specification is usually more under-specified, and the specifier does not want to restrict execution to those steps explicitly appearing in it. Therefore, for such specifications, synthesis is more appropriate. Moreover, synthesis of intermediate specifications may aid the specifier in identifying under-specified parts in the specification, and extending it accordingly. For the final specification, the choice between non-violating execution and synthesis depends on the amount of detail the specifier has introduced, and the level of under-specification in it. Even for the final specification the specifier may choose to leave certain parts under-specified and decide to use synthesis.

4   The Representation

4.1   The LSC Game Structure

As mentioned above, this work focuses on a subset of LSCs that includes only messages, and assumes that main charts include only system messages. We also assume that no LSC has multiple copies simultaneously open during runtime. Finally, all messages in the specification are assumed to be synchronous, i.e., the event of sending a message and receiving it are simultaneous.

Given an LSC specification, we construct a game structure G. Intuitively, the system controls a single variable, m_s, that represents the message sent by a system object in this step. The environment controls a variable m_e representing the message sent by an environment object in this step, and a set of variables

L that represent the current LSC configuration. In every turn, the system sends a single message or chooses not to (by using the special symbol $\perp$ for m'_s). The environment may choose an environment message to send only if the system did not send one of its own (i.e., if m'_s is $\perp$). The environment may also choose not to send any message (by using its own $\perp$ symbol for m'_e). The domain of m_e includes one additional special symbol, ∞ (may also be used only when m'_s is $\perp$). By using this symbol, the environment may force the game to stay forever in the current state. This forces the system to pass control back to the environment only at the end of a superstep, and is crucial for the correctness of the composition. Updating the configuration is deterministic, given m'_s and m'_e, and follows the semantics defined in [16] directly.

We adopt the superstep approach from [14]. Following a single environment step, the system may perform as many steps as it wishes (finitely many) in order to reach a state in which all LSCs are not active. Only then will the environment be allowed to play again.

More formally, given an LSC specification, we construct a game structure $G : \langle V, X, Y, \Theta, \rho_s, \rho_e, \varphi \rangle$ as follows:

- The set of variables $V = \{m_s, m_e\} \cup L$, as follows:
    - m_s represents the system message sent in this step. Its domain is the set of all messages sent by system objects in the specification, plus the symbol $\perp$, representing a no-op.
    - m_e represents the environment message sent in this step. Its domain is the set of all messages sent by environment objects in the specification, plus the symbols $\perp$ and ∞, both representing no-ops.
    - L contains the following:
        * For each lifeline i, a variable loc_i representing the location of the cut on lifeline i. Each location variable ranges over $0, \ldots, l^{max}$, where l^{max} is the last location of lifeline i.
        * For each LSC l, boolean variables $active_l$ and $hotViolated_l$ representing whether the LSC is active, and whether it was ever hot violated, respectively. We also introduce another boolean variable $prev_l$ for each LSC l, that represents the fact that in the previous timestep $active_l$ and $hotViolated_l$ were both false.
- $X = \{m_s\}$ is the only system variable.
- $Y = V \setminus X = \{m_e\} \cup L$ are the environment variables.
- The system transition relation is defined such that
  $\rho_s(m_s, m_e, L, m'_s) = 1 \iff [(m_s = \perp \wedge m_e = \infty \rightarrow m'_s = \perp)]$
- The environment transition relation is defined such that
  $\rho_e(m_s, m_e, L, m'_s, m'_e, L') = 1 \iff [(m'_s \neq \perp \rightarrow m'_e = \perp) \wedge$ the L' variables represent the state of the specification after sending m'_s and m'_e from state L]. We omit from this version of the paper the details of updating the L variables, as they are somewhat similar to those of [14], and are a direct translation of the operational semantics defined in [16].
- The winning condition is $\varphi = \square\lozenge(m_s = \perp \wedge prev_{l_1} \wedge \cdots \wedge prev_{l_k})$, where $l_1, \ldots, l_k$ are the LSCs in the specification. This represents the requirements

from the synthesized system, i.e., infinitely often the system must listen to environment events (this happens when $m_s = \bot$), and it must never violate the specification (represented by the requirements on the *prev* variables in φ, similarly to the requirement for ending a superstep in [14]). We denote by q the state formula in φ.
– The initial condition is $\Theta = [(m_s = m_e = \bot)\wedge$ values for L that represent all LSCs being closed].

For a variable u, we denote by $\bar{u}$ a valuation of u, and similarly for sets of variables.

4.2 Monotonicity

Given a realizable LSC game structure, the game structure corresponding to any subset of the LSCs is also realizable. Moreover, the restriction of a winning state in the composite structure to the subset one is a winning state in it. Intuitively, given a winning strategy for the composite specification, the same strategy can be used for the subset one. Since the strategy is winning for the composite specification, it satisfies the safety and liveness requirements of all LSCs in the entire specification, therefore it satisfies them for the LSCs in the subset one, and is a winning strategy for the subset specification.

Similarly, any extension of an unrealizable LSC game structure (by adding more LSCs) is also unrealizable.

5 The Synthesis Algorithm

We adapt the algorithm from [29] for games in which the system (controller) plays first in each turn. For lack of space, the details of this modification are omitted from this version of the paper. The result of the algorithm is a transition system $S = \langle V, \rho, \Theta \rangle$.

Definition 1. *Given a transition system $S = \langle V, \rho, \Theta \rangle$, the strategy induced by S is defined as: $f(s_0, s_1, \ldots, s_t) = \{m'_s | \exists m'_e, L' : (s_t, m'_s, m'_e, L') \models \rho\}$, i.e., the strategy allows any system message that appears in transitions from s_t.*
Since the induced strategy considers only the current state (state-strategy), we will use the short notation of $f(V)$.

6 Strategy Composition

Consider LSC game structures for two subset specifications G_1, G_2. The variables m_s and m_e are the only ones appearing in both. For now, assume the sets of messages appearing in the two specifications are equal, thus the domains of m_s and m_e are also equal. The case where some messages appear only in one of the specifications is discussed in Section 6.3. Clearly, $\rho_s^1 = \rho_s^2$ since they depend only on m_s, m_e and m'_s. ρ_e is the conjunction of ρ_e^1 and ρ_e^2 (each restricted to the variables relevant to it). q, the state formula in φ, is the conjunction of q_1 and q_2. Finally, the initial condition is $\Theta = \Theta_1 \wedge \Theta_2$. Define $G = \langle V, X, Y, \rho_s, \rho_e, \varphi, \Theta \rangle$ to be the LSC game structure for the composite specification.

6.1   The Composition Algorithm

We present an algorithm for the composition of transition systems that induce strategies. The algorithm has two main steps. It first computes the synchronous parallel composition of the transition systems, and then removes bad states from the result. A state is considered bad if it is a dead end, or will necessarily lead to one.

Given a transition system S inducing a strategy for the LSC game structure G, the following assertions, $Rlvt(V, m'_s)$ and $Self(V, m'_s)$, represent whether the message m'_s is relevant from a given state, and whether it leaves the LSC configuration unchanged from it, resp.

$$(s, \bar{m}'_s) \models Rlvt \Leftrightarrow \exists \bar{m}'_e, \bar{L}' : (s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho$$

$$(s, \bar{m}'_s) \models Self \Leftrightarrow \bar{m}'_s \neq \bot \wedge [\forall \bar{m}'_e, \bar{L}' : (s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho \rightarrow (s[L] = \bar{L}')]$$

Where $s[L]$ stands for the restriction of s to the variables of L.

Using these assertions, we define the operator *bad predecessor*, denoted $\ominus p$, as follows (where the notation $||q||$ stands for the set of states satisfying q):

$$|| \ominus p|| = \{s | \forall \bar{m}'_s[(s, \bar{m}'_s) \models Rlvt] \rightarrow$$
$$[(s, \bar{m}'_s) \models Self \vee \exists \bar{m}'_e, \bar{L}'((s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho \wedge (\bar{m}'_s, \bar{m}'_e, \bar{L}') \in ||p||)]\}$$

Thus, a state satisfies $\ominus p$ if any system message relevant from it is either a self message (i.e., it leaves the configuration unchanged), or opens an opportunity for the environment to get to a state satisfying p. By applying $\ominus p$ iteratively until a fixpoint is reached, we mark all bad states. By initially setting the set of bad states to $\emptyset$, dead-end states are marked as bad in the first iteration, and in following iterations, all states necessarily leading to them. We therefore define the set Bad as: $Bad = \mu B. \ominus B$, i.e., the minimal fixpoint of the bad predecessor predicate.

One can improve the performance of the fixpoint computation by first computing the set of reachable states in the transition system, and considering only those in the fixpoint iterations. Since the size of synthesized systems is typically significantly smaller than that of the whole specification model, the set of reachable states in them may be computed relatively easily.

The pseudo-code of an algorithm for the composition of synthesized systems is given in Fig. 2. The algorithm gets as input two transition systems, and if successful returns a new transition system. It computes the synchronous parallel composition of the two input systems, and removes bad states from it. If an initial state is found to be bad, then the algorithm terminates with no returned system. Otherwise, it constructs a transition relation that makes sure no bad states are ever reached. In the following sections we explore how this algorithm is either sound or complete, depending on the inputs.

6.2   Sound Composition

For the sound composition, we use the algorithm from Fig. 2 with synthesized systems as input. Intuitively, the algorithm tries to weave the two strategies in

```
1: procedure COMPOSE(System S1, System S2)
2:     S := S₁|||S₂
3:     bad ← calc_bad(S, reachable(S))
4:     if ∃s₀ ⊨ Θ, s₀ ∈ bad then return "Failed"
5:     else ρ(s, m'ₛ, m'ₑ, L') ←
6:         [ρ(s, m'ₛ, m'ₑ, L') ∧ (∀m̃'ₑ, L̃' : ρ(s, m'ₛ, m̃'ₑ, L̃') → (s, m'ₛ, m̃'ₑ, L̃') ∉ bad)]
7:         Return S
8:     end if
9: end procedure
```

Fig. 2. Pseudo-code for the composition algorithm

a way that does not violate either. If the strategies "agree" on the steps to be taken and their order, then the algorithm will succeed. Otherwise, the algorithm will fail. Note that the input strategies are not necessarily maximal, therefore the fact that the algorithm did not succeed in weaving them together does not mean there are no other winning strategies that can be successfully composed.

When the algorithm is given synthesized systems as input, it is sound, i.e., if it finds a system, then it is a synthesized system for the composite specification that induces a system-winning strategy. The formal proof of this claim is omitted from this version of the paper. Intuitively, we observe that if $m_s = \perp$ in a given state (i.e., the system decides to let the environment play), then the environment may use the ∞ symbol to force the system to stay in this state forever. Therefore, if $m_s = \perp$ in a system-winning state, then this state necessarily satisfies q. The soundness proof relies on this observation, along with the fact that the only variables shared between the subset systems are m_s and m_e. The proof considers the system found by the algorithm, and the strategy induced by it. It shows that any run compliant with it is necessarily compliant with the strategies for the subset specifications, and therefore winning in them. Then, following the observation above, it is also winning for the composite specification.

6.3   Augmented Strategies

Often when one considers composition of two specifications, there are messages that appear in one specification and not in the other. The synthesis algorithm as presented here will not allow steps not appearing explicitly in the specification. For composition, however, each part should be allowed to advance as much as it wishes, while using messages appearing only in it.

In this section, we show how a synthesized system may be augmented to allow steps that appear only in another (given) specification. Although the augmented system might not be a winning one anymore (it may now choose infinitely many steps from the other system without advancing), the composition algorithm is still sound when given these augmented systems as input.

Definition 2. *Given an LSC game structure, G, we define the set of system messages irrelevant to G to be the set of values for m'_s s.t. in any state, sending them changes nothing in the configuration, as follows:*

$$irrel(G) = \{\bar{m}'_s | \forall s, \bar{m}'_e, \bar{L}' : (s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho_e \rightarrow (\bar{L} = \bar{L}')\} \setminus \{\bot\}$$

Given two synthesized systems, S_1, S_2, for game structures G_1, G_2 resp., we create augmented systems $\tilde{S}_1, \tilde{S}_2$, by augmenting their transition relations with transitions in which the LSC configuration does not change, the system sends a message relevant only to the other system, and the environment sends no message, as follows (for $i, j \in \{1, 2\}, i \neq j$):

$$\tilde{\rho}_i(s, s') = \rho_i(s, s') \vee (s[L_i] = s'[L_i] \wedge s'[m_e] = \bot \wedge s'[m_s] \in irrel(G_i) \setminus irrel(G_j))$$

The algorithm, given augmented synthesized systems, is still sound, i.e., if it finds a system then it is a synthesized system for the composite specification that induces a winning strategy. The proof of the soundness is similar with augmented strategies, with the addition that steps resulting from augmenting one transition relation do not change the state of that system, and are always "real" steps in the other system, therefore there are finitely many such consecutive steps.

The strategy synthesized by the synthesis algorithm of [29] is one that allows only steps that strictly get it closer to a stable state (one that satisfies q). Following a stable state, any step leading to a winning state is allowed, and then again only steps that strictly get it closer to a stable state. One may further augment the system by allowing any step leading to a winning state from states that are not stable, but for which no "real" step has been taken since a stable state (but only ones resulting from augmenting the strategy).

6.4   Complete Composition

For the complete part, we use the same algorithm from Fig. 2 on inputs that represent an over-approximation of the maximal winning strategies. These will be termed *optimistic strategies*. We show that if the algorithm is given optimistic strategies as input, then it returns an optimistic strategy. Therefore, if no system is returned, then the composite specification is unrealizable.

Definition 3. *Given an LSC game structure, G, a strategy is optimistic if $\forall s$, and $\bar{m}'_s$: if $(\forall \bar{m}'_e, \bar{L}'[(s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho_e] \rightarrow (\bar{m}'_s, \bar{m}'_e, \bar{L}') \in win(G))$, then $\bar{m}'_s \in f(s)$. i.e., an optimistic strategy allows any system step that will necessarily lead to a winning state. A system inducing an optimistic strategy will be termed an optimistic system.*

In other words, an optimistic strategy must allow any transition to a state from which the system can win, and is therefore an over-approximation of the maximal winning strategy.

One can construct the minimal optimistic strategy by allowing any system step from a winning state after which any environment step reaches another winning state. Such a strategy will never violate any safety constraint (since it will not lead to a non-winning state), but it might violate liveness constraints.

If the composition algorithm from Fig. 2 is given optimistic systems as input, then it returns an optimistic system for the composite specification. Intuitively, the proof relies on the fact that a restriction of a winning state to a subset game structure is a winning state in it, therefore the synchronous parallel composition step induces an optimistic strategy. It then shows that no winning state is ever added to *Bad*, therefore no transitions between winning states are removed.

As a corollary, the algorithm is complete, i.e., if it finds no system, then the composite specification is unrealizable.

Note that even though the computed strategy is not the minimal optimistic strategy, it also will never violate a safety requirement, assuming the two input strategies never do so. This ensures that if we start from minimal optimistic strategies, and keep composing them (and the results of composing them), we will get a system that never causes a violation to any LSC. The reason the algorithm is not sound is that the resulting system, by being optimistic, does not guarantee to pass control back to the environment infinitely often. It might enter an infinite loop of system events, even though neither violates any chart.

6.5   Towards a Sound and Complete Algorithm

The composition algorithms described here are not meant to completely replace the full synthesis algorithm, but rather as fast alternatives that for some cases may work, and for others might fail. In such cases, there may be a need for applying full synthesis on the composite specification.

We now briefly describe possible extensions for these algorithms that together form a sound and complete algorithm. These are described in very general guidelines, and further implementation details are left as future work. The main reason for this is that the resulting algorithm may be too slow to be of any practical usage. Moreover, one part of the extension (the sound part) must be applied at the bottom-most level, i.e., when synthesizing a system. If one performs several composition steps, and now realizes he needs to further extend the system, he needs to start from the beginning, strengthen the synthesized systems and compose them together again. For the other part (the complete part), there does not seem to be a symbolic implementation.

The sound part of the algorithm can be extended as follows. The synthesis algorithm, as described above, finds strategies that at each step strictly move towards a stable state (one that satisfies q). By introducing an extra variable, *counter*, one can allow a given number of steps to move away from stable states (in each superstep). The intended usage is as follows: one sets *counter* to some low value, synthesizes his basic systems, and composes them. If some composition step fails, the basic systems can be resynthesized with larger initial *counter*, thus improving the chances for the compositions to succeed (yet extending its complexity and running time). If the composite system is realizable, then there exists a large enough initial *counter* for which the compositions will succeed.

The complete part can be extended by strengthening the computation of bad states. Currently, states are marked bad if they may lead only to themselves or to other bad states. However, the strategy might allow infinite loops of system
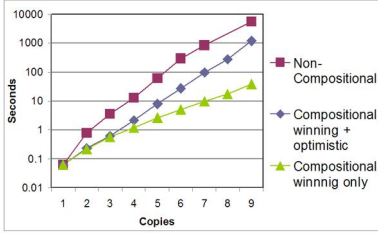
(a) Running time (log-scale) as a function of the number of disjoint copies of a 6-LSC specification.
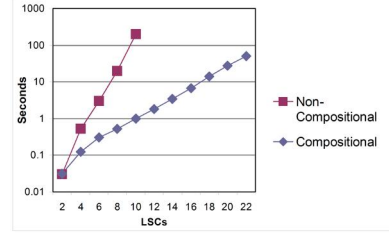
(b) Running time (log-scale) as a function of the number of LSCs, for inconsistent chain specifications.

Fig. 3. Running times for two parameterized examples

moves. Since the strategy does not violate any LSC (assuming initially minimal optimistic strategies were used), if such loops were avoided altogether, the resulting system would have been a winning one. Thus, by identifying such loops of increasing size, one improves his composed optimistic strategy, and if the system is unrealizable, eventually it will be proven as such.

An algorithm that alternatingly increases the counter and the loop size will be sound and complete. Details of these extensions are left as future work.

7   Results

We implemented our approach as part of the new Scenario-Based Tool [33] developed at Microsoft Research Cambridge, using TLV [31] for the symbolic computations. We now describe some experimental results from these tools.

The first example is a simple specification consisting of 6 LSCs, that refer to two objects. Following an initial environment message, the system must send the messages m1, m2, m2 in this order before passing control back to the environment. This specification was replicated, using disjoint sets of messages, with the number of replicas parameterized, and ranging from 1 to 10. Each copy was synthesized separately and the results were composed. Figure 3(a) compares the running time (log-scale) as a function of the number of copies, for: (a) the compositional approach, when only winning systems are composed, (b) the compositional approach, when both winning and optimistic systems are composed, and (c) non-compositional synthesis of the entire specification. The compositional approach, when only winning strategies are composed, is clearly significantly faster, but if a composition step would have generated an inconsistent specification, it could not have been proved without optimistic strategies.

Another example is adopted from [12], where it is shown that synthesis is strictly stronger than smart play-out. We modify the example to form a series of inconsistent specifications of growing lengths, where specification i requires considering i supersteps ahead in order to prove its inconsistency. Figure 3(b) shows the running time (log-scale) as a function of the specification size, for compositional synthesis (in which each LSC is synthesized separately and composed into the system) as

opposed to full synthesis of the specification. Clearly, the compositional approach saves significant running time. It is worth mentioning that inconsistency is proven when the composition is performed in a specific order. Different choices of the order did not manage to prove inconsistency.

Two more test cases were generated, both inspired by a biological model describing the process of vulval precursor cell fate determination in the development of the *C. elegans* nematode [20]. In one, (a simplification of) the different developmental steps were each synthesized separately, and the results were composed. This specification consists of 22 LSCs. Without composition, the synthesis of the entire system did not finish within 5 days, whereas the compositional approach obtains a running system in less than 3 minutes. The second specification focuses on the last developmental stage, and demonstrates the incremental nature of the specification process, while using compositional synthesis. This system was composed in 9.85 seconds, while the full non-compositional synthesis did not finish within 3 days. The latter example also acts as an example in which smart play-out may choose a superstep that is correct, but may lead to violations in future supersteps. The synthesized system, on the other hand, avoids such violations.

8   Related Work

In recent years there have been considerable research efforts on synthesizing executable systems from scenario-based requirements [26]. In many of these papers the requirements are given using a variant of classical Message Sequence Charts while the synthesized system is state-based. The main distinguishing feature of our work is that we consider synthesis from Live Sequence Charts, which are more expressive than most of the classical MSC variants.

One should realize that constructing a program from a specification is a long-known general and fundamental problem, dating back to work by Church [7] and tackled by [6,32]. There has also been much research on synthesis from a specification given in temporal logic, starting with closed systems, that do not interact with the environment [27,10], and later [30,1,36] dealing with the synthesis of open systems from Linear Temporal Logic specifications. The problems of realizability checking and synthesis from LTL are shown to be 2EXPTIME-complete. Despite this high complexity, progress has been made in the development and application of synthesis algorithms, by proposing new algorithms [25], using heuristic approaches [11], considering subsets of temporal logic [2,28], smart implementation and application [19,3,34]. A compositional method for synthesis is presented in [24] building upon basic results first described in [25].

Synthesis from LSCs was first studied in [13], and is tackled there by defining consistency, showing that an entire LSC specification is consistent if and only if it is satisfiable by a state-based object system, and then synthesizing a satisfying system. The work in [13] considers a core LSC subset consisting of messages only, similar to this paper, but does not implement the algorithms or study the practical questions related to implementation. A game theoretic approach to synthesis from LSCs involving a reduction to parity games is described in [4],

the authors summarize the experimental results as negative, partially due to a poor prototype implementation. Synthesis from LSCs using a reduction to CSP is described in [35]. In [22] synthesis from LSC is tackled somewhat similarly to this paper, however compositional synthesis is not considered at all.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and Unrealizable Concurrent Program Specifications. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
2. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller Synthesis for Timed Automata. In: IFAC Symp. on System Structure and Control, pp. 469–474 (1998)
3. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic Hardware Synthesis from Specifications: A Case Study. In: Proceedings of the Design, Automation and Test in Europe, pp. 1188–1193 (2007)
4. Bontemps, Y., Heymans, P., Schobbens, P.Y.: From Live Sequence Charts to State Machines and Back: A Guided Tour. IEEE Trans. Software Eng. 31(12), 999–1014 (2005)
5. Bontemps, Y., Schobbens, P.: Synthesizing Open Reactive Systems from Scenario-Based Specifications. In: Proc. of the 3rd Int. Conf. on Application of Concurrency to System Design (ACSD 2003) (2003)
6. Büchi, J., Landweber, L.: Solving Sequential Conditions by Finite-State Strategies. Trans. Amer. Math. Soc. 138, 295–311 (1969)
7. Church, A.: Logic, Arithmetic and Automata. In: Proc. 1962 Int. Congr. Math, Upsala, pp. 23–25 (1963)
8. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. J. on Form. Meth. in Sys. Design 19(1), 45–80 (2001)
9. Damm, W., Toben, T., Westphal, B.: On the Expressive Power of Live Sequence Charts. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 225–246. Springer, Heidelberg (2007)
10. Emerson, E., Clarke, E.: Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. Science of Computer Programming 2, 241–266 (1982)
11. Harding, A., Ryan, M., Schobbens, P.: A New Algorithm for Strategy Synthesis in LTL Games. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 477–492. Springer, Heidelberg (2005)
12. Harel, D., Kantor, A., Maoz, S.: On the Power of Play-Out for Scenario-Based Programs (to appear, 2009)
13. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. Int. J. of Found. of Comp. Sci (IJFCS) 13(1), 5–51 (2002)
14. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of Behavioral Requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)
15. Harel, D., Kugler, H., Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
16. Harel, D., Marelly, R.: Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine (2003)
17. Harel, D., Segall, I.: Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 485–499. Springer, Heidelberg (2007)

18. ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report (1996)
19. Jobstmann, B., Bloem, R.: Optimizations for LTL Synthesis. In: 6th Conf. Formal Methods in Computer Aided Design (FMCAD 2006) (2006)
20. Kam, N., Kugler, H., Marelly, R., Appleby, L., Fisher, J., Pnueli, A., Harel, D., Stern, M., Hubbard, E.: A Scenario-Based Approach to Modeling Development: A Prototype Model of C. Elegans Vulval Fate Specification. Developmental Biology 323(1), 1–5 (2008)
21. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Scenario-Based Specifications. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 445–460. Springer, Heidelberg (2005)
22. Kugler, H., Plock, C., Pnueli, A.: Controller Synthesis from LSC Requirements. In: 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009). LNCS. Springer, Heidelberg (2009)
23. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. Technical report, Microsoft Research (2009)
24. Kupferman, O., Piterman, N., Vardi, M.: Safraless Compositional Synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
25. Kupferman, O., Vardi, M.: Safraless Decision Procedures. In: Proc. 46th IEEE Symp. on Found. of Computer Science, Pittsburgh, October 2005, pp. 531–540 (2005)
26. Liang, H., Dingel, J., Diskin, Z.: A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In: Proc. of the Intl. Work. on Scenarios and State Machines: Models, Algs., and Tools (SCESM 2006), pp. 5–12 (2006)
27. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. ACM Trans. Programming Languages and Systems 2, 90–121 (1980)
28. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
29. Pnueli, A.: Extracting Controllers for Timed Automata. Technical report, NYU (2005)
30. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proc. 16th ACM Symp. Princ. of Prog. Lang, pp. 179–190 (1989)
31. Pnueli, A., Shahar, E.: A Platform for Combining Deductive with Algorithmic Verification. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 184–195. Springer, Heidelberg (1996)
32. Rabin, M.: Decidability of Second Order Theories and Automata on Infinite Trees. Trans. Amer. Math. Soc. 141, 1–35 (1969)
33. Microsoft Research Cambridge, Scenario-Based Tool for Biological Modeling (2009), http://research.microsoft.com/SBT/
34. Sohail, S., Somenzi, F., Ravi, K.: A Hybrid Algorithm for LTL Games. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 309–323. Springer, Heidelberg (2008)
35. Sun, J., Dong, J.S.: Synthesis of Distributed Processes from Scenario-Based Specifications. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 415–431. Springer, Heidelberg (2005)
36. Wong-Toi, H., Dill, D.: Synthesizing Processes and Schedulers from Temporal Specifications. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 272–281. Springer, Heidelberg (1991)

Computing Weakest Strategies for Safety Games of Imperfect Information

Wouter Kuijper and Jaco van de Pol

University of Twente[*]
Formal Methods and Tools
Dept. of EEMCS
{W.Kuijper,J.C.vandePol}@ewi.utwente.nl

Abstract. CEDAR (Counter Example Driven Antichain Refinement) is a new symbolic algorithm for computing weakest strategies for safety games of imperfect information. The algorithm computes a fixed point over the lattice of *contravariant antichains*. Here contravariant antichains are antichains over *pairs* consisting of an information set and an allow set representing the associated move. We demonstrate how the richer structure of contravariant antichains for representing antitone functions, as opposed to standard antichains for representing sets of downward closed sets, allows CEDAR to apply a significantly less complex controllable predecessor step than previous algorithms.

1 Introduction

Many problems related to synthesis and verification of systems reduce naturally to solving games [13,8]. In particular, games of imperfect information form a class of games where the players have only partial access to the current state, which leads to the players having imperfect information about the game's exact location. In our field this class of games is important since the concept of partial observability seems to arise quite naturally in several applications.

For example, the Controller Synthesis problem requires a control strategy for a plant to be automatically synthesized [9,10]. Here, in general, not everything about the plant will be directly observable for the controller. On a more fine grained level of control, the problem of Motion Planning with Uncertainty naturally exhibits aspects of imperfect information. Finally, for a general problem like Interface Compatibility Checking [5] on componentized, object-oriented programs, we may consider the private fields and methods of an object leading to internal, non-observable behaviour.

When we restrict ourselves to safety objectives but maintain partial observability we are dealing with safety games of imperfect information. For this class, if a game is solvable, there always exists a *weakest* solution which subsumes all other solutions to the same game.

Complexity. It is known that partial observability bumps the complexity of two player games with perfect observation to a higher class [11]: the exponential complexity for

---

[*] This work was partly funded by NWO project 600.065.120.24N20.

solving games of imperfect information derives from an inherent subset construction needed to analyze the information sets of the player.

Recently, however, progress has been made in the form of symbolic algorithms that are able to analyze nondeterministic automata while avoiding the explicit subset construction for determinization [14,15,6]. One such new class of algorithms works by computing fixed points over antichains which are sets of pairwise incomparable sets of states (information sets). Although the inherent complexity of the problem still remains exponential, on instances the authors report significant efficiency gains [2].

Contribution. In our approach we limit the scope to safety objectives which allows for a symbolic algorithm that can compute weakest strategies. This is complementary to the approach of [4]. Our approach is useful for cases where (1) everything that one wants to synthesize is expressible as a safety property (e.g.: hard timeliness constraints for instance are expressible as a safety property) and/or (2) the result must be reusable and amenable to further analysis/composition/optimization. It is especially in case (2) where weakest strategies really shine.

Since a weakest strategy for a given game subsumes all possible safe strategies it is useful as a *safety monitor* (i.e: for supervising software or users that cannot be completely guaranteed safe). Computing the weakest strategy may also form part of a preprocessing step for generating a safe input graph to a second synthesis procedure that can optimize some performance measure that is not expressible as a safety property. Finally, weakest safety strategies are useful in a compositional setting where the behaviour of the context is not known beforehand so that a most general solution is necessary in order not to exclude possible safe compositions with a concrete context.

Our main contribution is a new algorithm named CEDAR (Counter Example Driven Antichain Refinement). In a nutshell, the algorithm computes a fixed point over an enriched form of antichains which we call *contravariant antichains*. Contravariant Antichains enjoy most of the properties of normal antichains, but they can represent knowledge based strategies, which are antitone functions from information sets to allow sets, rather than just sets of downward closed information sets. This additional structure allows us to symbolically compute, not just the set of winning initial information sets, but the entire weakest knowledge based strategy.

As a second contribution our approach permits to significantly simplify the controllable predecessor step. In contrast to [4] we only treat a single counterexample observation to the observation-closedness condition for the contravariant antichain, as opposed to treating all counterexample observations at every iteration.

Related Work. A result that contrasts with the symbolic approach is [3]. Here games are solved by searching the knowledge based subset construction in a forward direction (starting from the initial information set). The winning strategy is constructed while traversing this graph using an efficient on-the-fly fixed point algorithm due to [7]. This means that loosing states are pruned out and back propagated at an early stage but it does not constitute a fully symbolic algorithm since the algorithm still explicitly constructs (a subgraph of the) knowledge based subset construction.

The algorithm presented in [4] works for omega regular winning conditions, which from one point of view makes it more general than CEDAR, which works only for safety

objectives. However this generality also comes at a price since for omega regular objectives there is in general no *weakest* strategy [1]. Indeed the algorithm computes the set of winning information sets, i.e.: the weakest information sets from which there still exists *a* winning strategy.

Recently the same authors show that the antichain representation of the largest winning regions for a given parity game does not allow to recover the winning strategy directly [2]. The authors present an algorithm that can construct a winning strategy using antichains as the underlying datastructure for representing sets of downward closed state sets. Clearly, since they are dealing with parity games, the algorithm will construct a strategy that is not necessarily the weakest. This approach can be seen as complementary to ours. Our approach is limited to safety games, but computes the strategy directly in the form of a contravariant antichain, and ensures that the resulting strategy is the weakest.

Structure of the paper. The paper is structured as follows. In Section 2 we define and discuss imperfect information safety games, strategies, and weakest strategies. In Section 3 we introduce contravariant antichains which is the new datastructure underlying CEDAR. In Section 4 we present the CEDAR algorithm. And finally in Section 5 we give preliminary experimental results and concluding remarks.

2   Safety Games of Imperfect Information

In this section we introduce formally the notion of a *safety game of imperfect information*, and we define the *weakest, antitone knowledge based strategy* for a given game.

Definition 1 (Safety Games). A *safety game of imperfect information* G is a tuple

$$G = (L, C^{\text{out}}, C^{\text{in}}, \alpha, \beta, \delta, i^{\text{init}})$$

consisting of a finite set of *game locations* L, a finite set of *control outputs* C^{out}, a finite set of *control inputs* C^{in}, an *output labeling* $\alpha : L \to C^{\text{out}}$, an *input labeling* $\beta : L \to C^{\text{in}}$, a *game board* $\delta \subseteq L \times L$, and a set of *initial locations* $i^{\text{init}} \subseteq L$ (also called the *initial information set*). We define $O = C^{\text{out}} \times C^{\text{in}}$ as the set of *observations*, an observation $o \in O$ is written as $o = c^{\text{out}}/c^{\text{in}}$. As a convenience we define labeling $\gamma : L \to O$ such that $\gamma(\ell) = \alpha(\ell)/\beta(\ell)$. We define $A = 2^{C^{\text{out}}}$ as the set of *allow sets*. Let $\alpha^{-1}(a) = \{\ell \in L \mid \exists c^{\text{out}} \in a.\alpha(\ell) = c^{\text{out}}\}$, and $\gamma^{-1}(o) = \{\ell \in L \mid \gamma(\ell) = o\}$; since it is always clear from the context where a set of locations is required, throughout the paper we will leave the conversions $\alpha^{-1}(\cdot)$ and $\gamma^{-1}(\cdot)$ implicit.   ◁

A safety game of imperfect information should be interpreted as a game between two players: *the safety player* and *the reachability player*. The objective for the safety player is to keep the game running forever. The objective for the reachability player is to reach a deadlock state, i.e.: a location ℓ in which it holds $\delta(\ell) = \varnothing$.

Since we aim for a framework where strategies are ordered with respect to permissiveness we introduce moves for the safety player as *allow sets*. In this way we can have a subsumption relation on the moves for the safety player. The moves for the reachability player are then the *concrete successor locations* that are allowed by the game board

and by the allow set chosen by the safety player. For example, if we are in game location $\ell \in L$ and the safety player chooses move $a \in A$, the reachability player must choose a successor location from the *forcing set* which is defined as $\delta(\ell) \cap a$. It is up to the safety player to ensure that her forcing set never becomes empty. Below we illustrate the definition with a concrete example of a safety game.

Example 1 (Pennymatching). We introduce a simple game of *penny-matching*. In this game, at each round, both players choose a side to a penny. If the safety player forfeits her choice by playing $a = \{h, t\}$ (heads *or* tails) the reachability player will choose for her. This may seem counterintuitive on this simple example, however note that, from a control perspective, this is a reasonable model: when the safety player permits two possible, distinct control outputs and the game-board does not resolve this choice either, she automatically yields this forcing power to her opponent.

The rules of the game are now as follows: if both players play heads the game is over and it is a win for the reachability player, in all other cases the game simply continues. To make the game slightly more interesting we stipulate that the reachability player cannot surprise the safety player by playing heads twice in a row.

Finally, in order to investigate the effect of imperfect information, we introduce two variants of the game: *open* pennymatching where the safety player can observe the coin of the reachability player, and, the harder variant, *blind* pennymatching where the safety player has no information about what side the reachability player chooses at each turn. According to definition 1 we may model these games as follows:

$L_{\text{penny}} = \{h, t\} \times \{h, t\}$	$C_{\text{open}}^{\text{in}} = \{h, t\}$	$C_{\text{blind}}^{\text{in}} = \{\mathbf{x}\}$	$C_{\text{penny}}^{\text{out}} = \{h, t\}$
$i_{\text{penny}}^{\text{init}} = \{ht\}$	$\beta_{\text{open}}(sr) = r$	$\beta_{\text{blind}}(sr) = \mathbf{x}$	$\alpha_{\text{penny}}(sr) = s$
$\delta_{\text{penny}} = \{(sr, s'r') \in L \times L \mid \neg(s = r = h) \wedge (r = h \rightarrow r' \neq h)\}$			

Note that we consistently shorten a location $(s, r) \in L_{\text{penny}}$ as a juxtaposition sr. In Figure 1 we show a fragment of the unraveling of this game into a *game dag*. The intermediate forcing sets (where the reachability player will choose his move) are shown as dotted boxes. Note that the move $a = \{h, t\}$ (the weakest move for the safety player) played from the initial game state transitively leads to all four possible game locations including the deadlock at 'hh'. Further note that, played from the location 'th' the same move transitively leads to 'ht' or 'tt' which are both still safe.

The dashed lines connecting two nodes of the game dag indicate that for the *blind* version of the game these states in the unraveling of the game are *indistinguishable* for the safety player. Note that for the *open* version of the game it is immediately clear what would be the rational strategy for the safety player, the safety player just has to avoid the deadlock state marked with $\times$ hence she has to play $a = \{t\}$ all the time until she observes 't/h' after which she may relax her move to $a = \{h, t\}$. For the *blind* version of the game this is not so straightforward since her observation $t/\mathbf{x}$ cannot distinguish between the successor locations th and tt, and for that reason she can never know for sure to be in location th. How this is analyzed formally is shown in Example 2.   ◁

Knowledge Based Subset Construction. So far we have not explicitly dealt with the fact that the safety player has only a limited number of observations at her disposal. The
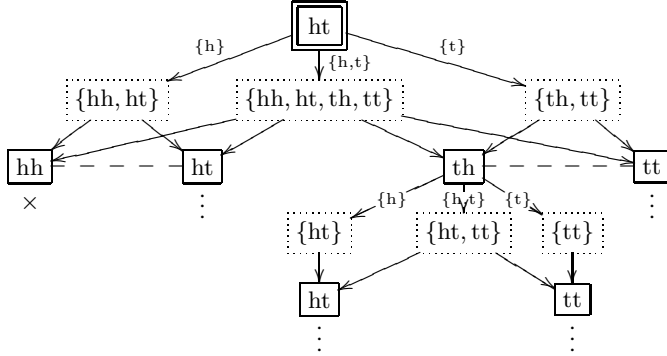
Fig. 1. Game DAG for the pennymatching game of Example 1

fact that the safety player can only make a limited observation of the current state is commonly referred to as *partial observability*. Partial observability leads to the safety player having only *imperfect information* about the exact location of the game. The impact of imperfect information in the analysis of games is huge due to the fact that the game graph δ is not *observation deterministic*. This means that distinct branchings in δ are not always distinguishable for the safety player. It is well known that this type of non-determinism can be resolved by applying a subset construction. We give the definitions below. Recall that, for a given location, $\gamma(\ell) = o$ denotes the observable information on ℓ, in this sense the set of observations O partitions L.

Definition 2 (Knowledge Based Subset Construction). For a given game, with I we denote the set of *information sets* defined as $I = 2^L$, and with Δ we denote the *knowledge based subset construction* which is defined as a graph over information sets $\Delta \subseteq I \times I$ as follows:

$$\Delta = \{(i, i') \in I \times I \mid \exists o \in O.i' = \delta(i) \cap o\}$$

Note that the image of δ on i is $\delta(i) = \bigcup_{\ell \in i} \delta(\ell)$, now $i' = \delta(i) \cap o$ represents the strongest knowledge the safety player has about the successor location upon observing o with knowledge i about the source location.   ◁

Example 2 (Knowledge Based Subset Construction). Figure 2 shows a fragment of the knowledge based subset construction for the *blind* version of the pennymatching game from Example 1. We do not normally draw the empty information set, we do include the intermediate forcing sets again for clarity, and now, in addition, we also show the observations that result from the moves for the reachability player.

From this graph it is clear what is the rational strategy for the blind pennymatching game: always play $a = \{t\}$ to avoid information sets that include a deadlock state.   ◁

Knowledge Based Strategies. We are now in a position to introduce the concept of a strategy for the safety player. This definition also determines the winning condition: the safety player wins the game iff she has a strategy to force an infinite play.
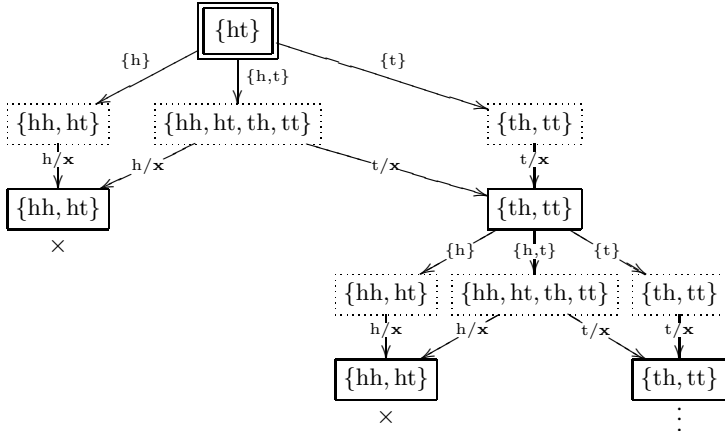
Fig. 2. Knowledge based subset construction for the blind pennymatching game

Definition 3. For a given game, a *knowledge based strategy* is a function $f : I \to A$. With F we denote the set of all knowledge based strategies. For a given strategy $f \in F$ and information set $i_0 \in I$, with $\mathsf{outcome}(G, f, i_0)$ we denote the *outcome of f on G starting from i_0* as a set of non-empty traces of game locations annotated with information states: $\mathsf{outcome}(G, f, i_0) \subseteq (L \times I)^+$. This is defined as follows:

$$\mathsf{outcome}(G, f, i_0) = \{\ell_0 i_0 \ldots \ell_n i_n \mid \forall j \leq n.\ell_j \in i_j, \text{ and } \forall j < n.$$
$$(\ell_j, \ell_{j+1}) \in \delta \text{ and } (i_j, i_{j+1}) \in \Delta \text{ and } \alpha(\ell_{j+1}) \in f(i_j)\}$$

These are all possible finite (partial) plays that may arise when our safety player is playing according to knowledge based strategy f. An outcome is *safe* iff no play ends in a deadlock (every finite play has a proper extension). We say that a strategy f is *safe* for G iff for all $i \in I$ either $\mathsf{outcome}(G, f, i)$ is safe, or $f(i) = \varnothing$. A strategy is *winning* iff it is safe and $f(i^{\mathrm{init}}) \neq \varnothing$. A game is *solvable* iff it permits a winning strategy.   ◁

An Inductive Definition of Safety. For the exposition of CEDAR we need to give an equivalent, inductive characterization of safety in terms of the following two elementary properties of knowledge based strategies. The first property is *obstinacy*, intuitively a strategy is obstinate if it blocks completely on information sets for which an empty forcing set is possible, or, equivalently, it returns a non-empty allow set only if each of the states in the information set has at least one valid successor in the underlying game board intersected with the allow set. The second property is *observation-closedness*, intuitively a strategy is observation-closed if it can guarantee that non-blocking states will, for every possible observation, lead to non-blocking successor states.

One may think of these two properties as an inductive definition of safety where obstinacy forms the base case, and observation-closedness forms the inductive case.

Definition 4 (Inductive Safety). For a given game, a knowledge based strategy $f \in F$ is *obstinate* iff for all $i \in I$ such that there exists $\ell \in i$ for which $\delta(\ell) \cap f(i) = \varnothing$ it holds $f(i) = \varnothing$. A knowledge based strategy $f \in F$ is *observation-closed* iff for all $i \in I$ and $o \in O$ such that $\delta(i) \cap f(i) \cap o \neq \varnothing$ it holds that $f(\delta(i) \cap o) \neq \varnothing$.   ◁

The following lemma establishes that *obstinacy* and *observation-closedness* are sufficient conditions to characterize safety for knowledge based strategies.

Lemma 1 (Inductive Safety). For a given game, a strategy is safe iff it is both obstinate and observation-closed. $\lhd$

Weakest Strategies. In the previous sections we have consistently defined *a* solution to a safety game as *any* winning strategy. In this section we sharpen this to *the weakest*, or *most permissive* winning strategy. Intuitively, a winning strategy is the most permissive winning strategy if for all plays the strategy always yields the largest possible allow set that is sufficient for keeping the future play safe. Formally, this means we introduce an ordering on F with respect to which we may select the greatest element in the subset of safe strategies.

Definition 5. For a given game, we define a weak partial order $\sqsupseteq$ on F such that $f' \sqsupseteq f$ iff for all $i \in I$ it holds $f'(i) \supseteq f(i)$. We say f' is *weaker* or *more permissive* than f. A strategy $f \in F$ is *antitone* iff for all $i, i' \in I$ it holds: $i \subseteq i'$ implies $f(i) \supseteq f(i')$. $\lhd$

We first show that for obtaining weakest, safe strategies, we can restrict our attention to antitone strategies.

Lemma 2. For a given game, for any safe strategy f there exists a weakest, safe, antitone strategy f' such that $f' \sqsupseteq f$. $\lhd$

Proof. Given a strategy f that is obstinate and observation-closed, we can define $g(i) := \bigcup\{f(i') \mid i \subseteq i'\}$. It is straightforward to show that $g \sqsupseteq f$, and g is antitone, obstinate, and observation-closed.

Given any two antitone, obstinate and observation-closed strategies f_1 and f_2, it can be checked that their join, defined as $(f_1 \sqcup f_2)(i) := f_1(i) \cup f_2(i)$ is also antitone, obstinate, and observation-closed. Hence, as the lattice of antitone safe strategies is finite, it is a complete lattice, and the weakest safe antitone $f' \sqsupseteq g$ exists. $\square$

We can summarize this discussion by the following definition and theorem:

Definition 6. With f_G we denote the weakest, safe, antitone strategy on game G. $\lhd$

Theorem 1. For any game G it holds that G is solvable iff $f_G(i^{\mathrm{init}}) \neq \varnothing$. $\lhd$

3 A Datastructure for Representing Antitone Functions

In this section we develop an efficient, symbolic representation for antitone functions as *contravariant antichains* which are antichains over domain/codomain pairs. First we give the general definition of a contravariant antichain, next we instantiate this definition and use it as the main datastructure underlying CEDAR. As it turns out, contravariant antichains are suitable for representing both knowledge based strategies as well as the set of open counterexample observations.

Definition 7 (Contravariant Antichains). Let $(S, \subseteq^{\mathrm{s}})$ be some finite, partially ordered domain that forms a complete lattice, and $(T, \subseteq^{\mathrm{t}})$ some finite, partially ordered

co-domain that forms a complete lattice. A *contravariant relation* is a set $h \subseteq S \times T$ that represents a function $\llbracket h \rrbracket : S \rightarrow T$ such that $\llbracket h \rrbracket(s) = \bigcup \{ t' \mid \langle s', t' \rangle \in h, s \subseteq^s s' \}$, i.e.: an element s from the domain S is *implicitly* mapped to an element t of the codomain T that is the join of all t' to which weaker s' than s are *explicitly* related in h. We will frequently abuse notation and write simply $h(s)$ instead of $\llbracket h \rrbracket(s)$. We define the weak partial order $\subseteq^{\langle s,t \rangle}$ on $S \times T$ as the product order: $\langle s, t \rangle \subseteq^{\langle s,t \rangle} \langle s', t' \rangle$ iff $s \subseteq^s s'$ and $t \subseteq^t t'$. The corresponding strict partial order is denoted by $\subset^{\langle s,t \rangle}$.

A *contravariant antichain* is a contravariant relation consisting of pairwise $\subset^{\langle s,t \rangle}$-incomparable domain/codomain pairs. With $\mathcal{C}[S, T]$ we denote the set of contravariant antichains from S to T. $\triangleleft$

Below we give an example of the use of contravariant antichains for representing knowledge based strategies.

Example 3 (Strategies as Contravariant Antichains). Assuming the definition in example 1, the following contravariant antichain $h_{\mathrm{penny}} \in \mathcal{C}[I, A]$ represents a knowledge based strategy for the pennymatching game:

$$h_{\mathrm{penny}} = \{ \langle \{\mathrm{ht}, \mathrm{tt}, \mathrm{th}\}, \{\mathrm{t}\} \rangle, \langle \{\mathrm{th}\}, \{\mathrm{h}, \mathrm{t}\} \rangle \}$$

Note that $\llbracket h_{\mathrm{penny}} \rrbracket$ is a winning strategy for the (blind) pennymatching game. For this specific instantiation of Definition 7 we will refer to the domain/codomain pairs as *info/allow* pairs.

It is clear that contravariant antichains with their semantics in the domain of antitone functions form an adequate representation of knowledge based strategies. They are, however, not *canonical*. To see this note that the following contravariant antichain k_{penny} is equivalent to h_{penny} in the sense that they represent the same strategy:

$$k_{\mathrm{penny}} = \{ \langle \{\mathrm{ht}, \mathrm{tt}, \mathrm{th}\}, \{\mathrm{t}\} \rangle, \langle \{\mathrm{th}\}, \{\mathrm{h}\} \rangle \}$$

Note that $\llbracket k_{\mathrm{penny}} \rrbracket(\{\mathrm{th}\}) = \{\mathrm{h}\} \cup \{\mathrm{t}\} = \{\mathrm{h}, \mathrm{t}\}$. $\triangleleft$

Apparently there is still structure in a contravariant antichain. To characterize it, we lift $\subseteq^s$ and $\subseteq^t$ to *preorders* on $S \times T$, by defining $\langle s, t \rangle \subseteq^{\langle s,\cdot \rangle} \langle s', t' \rangle$ iff $s \subseteq^s s'$, and similar for $\subseteq^{\langle \cdot,t \rangle}$. Note that for the corresponding strict partial orders, we have: $\subset^{\langle s,t \rangle} = (\subset^{\langle s,\cdot \rangle} \cap \subseteq^{\langle \cdot,t \rangle}) \cup (\subseteq^{\langle s,\cdot \rangle} \cap \subset^{\langle \cdot,t \rangle})$.

We now propose two possible canonical classes of contravariant antichains called *saturated contravariant antichains* and *sparse contravariant antichains,* respectively. A contravariant antichain is called *saturated* if it contains all $\subseteq^{\langle s,t \rangle}$-maximal domain/codomain pairs in the graph of the antitone function it represents. The strategy h_{penny} from example 3 is saturated. A contravariant antichain is *sparse* if it contains only $\subseteq^{\langle s,\cdot \rangle}$-principal pairs, which are all pairs for which the target is disjoint from the joined targets of all domain/codomain pairs that have a weaker source element. The strategy k_{penny} from example 3 is sparse. Both on saturated and sparse instances, $\subseteq^{\langle s,\cdot \rangle}$ is antisymmetric.

A contravariant antichain in its sparse normal form is generally smaller because it only contains pairs that have "added value". However, in principle, it carries the same information as a contravariant antichain in its saturated normal form. In Section 4 we show how both normal forms are useful in practice.

Definition 8 (Sparse Contravariant Antichains). Let $(S, \subseteq^s)$, $(T, \subseteq^t)$ be complete, finite lattices. For a given contravariant relation $h \subseteq S \times T$ and $s \in S$ with $h{\uparrow}s$ we denote h *above* s, defined as $h{\uparrow}s = \{\langle s', t' \rangle \in h \mid s \subset^s s'\}$. With $\mathsf{S}(h)$ we denote the *source set* of h defined as $\mathsf{S}(h) = \{s \in S \mid \exists t. \langle s, t \rangle \in h\}$. With $\lfloor h \rfloor$ we denote the *sparse normal form of h*. This is defined as follows:

$$\lfloor h \rfloor = \{\langle s, t \rangle \mid s \in \mathsf{S}(h) \text{ and } t = [\![h]\!](s) \setminus [\![h{\uparrow}s]\!](s) \text{ and } t \neq \varnothing\}$$

We say h is *sparse* iff $h = \lfloor h \rfloor$. With $\lfloor \mathcal{C} \rfloor [S, T]$ we denote the set of all sparse contravariant antichains from S to T.   ◁

Definition 9 (Saturated Contravariant Antichains). Given any contravariant relation $h \subseteq S \times T$, with $\lceil h \rceil$ we denote the *restriction of h to $\subseteq^{\langle s,t \rangle}$-maximal elements*. This is defined as follows:

$$\lceil h \rceil = \{\langle s, t \rangle \in h \mid t \neq \varnothing \text{ and } \nexists \langle s', t' \rangle \in h. \langle s, t \rangle \subset^{\langle s,t \rangle} \langle s', t' \rangle\}$$

We define the *contravariant closure* as follows:

$$[\![h]\!] = \lceil \{\langle s, t \rangle \mid \exists \langle s_1, t_1 \rangle, \dots, \langle s_m, t_m \rangle \in h. s = \cap_{1 \leq j \leq m} s_j \text{ and } t = \cup_{1 \leq j \leq m} t_j\} \rceil$$

i.e.: for any non-empty subset of domain/codomain pairs we take the meet of the source elements and the join of the target elements. We say h is *saturated* iff $h = [\![h]\!]$, with $[\![\mathcal{C}]\!][S, T]$ we denote the set of all saturated contravariant antichains from S to T. For $h, k \in [\![\mathcal{C}]\!][I, A]$ we let $h \sqcup k$ be the *join* of h and k, defined as $h \sqcup k = [\![h \cup k]\!]$.   ◁

4   An Algorithm for Computing Weakest Strategies

Algorithm 1 computes the weakest, safe knowledge based strategy for a given safety game of imperfect information. The algorithm works by approximating from above an obstinate, observation-closed fixed point in the lattice of saturated contravariant antichains. We recall our characterization of safety in terms of obstinacy and observation-closedness in Definition 4. The algorithm is based on the fact that we can maintain *obstinacy* as an invariant by never allowing any source location with empty forcing sets into the strategy. *Observation-closedness* then remains as the fixed point condition that the algorithm needs to work towards. The idea is to approach the fixed point by treating, at each iteration, a counterexample against observation-closedness. A counterexample against observation-closedness consists of an information set $i \in I$ and an observation $o \in O$, such that $\delta(i) \cap f(i) \cap o \neq \varnothing$ and $f(\delta(i) \cap o) = \varnothing$, i.e., o can actually be observed, but the strategy blocks on the resulting information set.

In order to avoid an explicit iteration over all possible observations, CEDAR computes, for a given $i \in I$, the set of counterexample observations *symbolically*. To show how this is done we now give an alternative characterization of the set of counterexample observations as the set of *unexplained observations*.

Intuitively, an observation $o \in O$ from an information set $i \in I$ is *explained* by another information set $i'' \in I$, if the successor information set $i' = \delta(i) \cap o$ is a subset of i'' and $f(i'') \neq \varnothing$. Note that, since f is antitone, it follows that $f(i') \neq \varnothing$ hence (i, o) is *not* a counterexample to observation-closedness. The set of observations that are *not explained* by any i'' can now be computed symbolically as

$$O^\dagger = \bigcap_{i'' \in I. f(i'') \neq \varnothing} \gamma((\delta(i) \cap f(i)) \setminus i'') \tag{1}$$

That is: $O^\dagger$ contains all observations for which the successor information set from i is not completely inside any suitable i''. In Definition 10, we see how the contravariant antichain representation of strategies simplifies the intersection in Equation 1 further.

Prerequisite Functions. Before we discuss CEDAR in more detail we first define the three helper functions in terms of which the algorithm is expressed.

Definition 10 (Unexplained Observations). For $h \in [\![\mathcal{C}]\!][I, A]$ we let $\widehat{h}$ be the antichain of maximal information sets in h, defined as: $\widehat{h} = \{i'' \in \mathsf{S}(h) \mid \nexists i' \in \mathsf{S}(h).i'' \subset i'\}$. We let $\mathsf{Uobs}(h) \in \lfloor \mathcal{C} \rfloor[I, 2^O]$ be the set of *unexplained observations*, defined as follows:

$$\mathsf{Uobs}(h) = \lfloor \{\langle i, O^\dagger \rangle \mid i \in \mathsf{S}(h) \text{ and } O^\dagger = \bigcap_{i'' \in \widehat{h}} \gamma((\delta(i) \cap h(i)) \setminus i'')\} \rfloor \tag{2}$$

$\triangleleft$

Note the restriction to maximal information sets in Equation 2. This is valid since an observation that is not explained by any of the maximal information sets will certainly not be explained by any of the weaker information sets. Below we give an example of how defining Equation 2 is used to compute the set of counterexample observations.

Example 4 (Unexplained Observations for Pennymatching). We let $h \in [\![\mathcal{C}]\!][I, A]$ be $h = \{\langle\{\mathrm{ht}, \mathrm{tt}, \mathrm{th}\}, \{\mathrm{h}, \mathrm{t}\}\rangle\}$, this is the weakest obstinate strategy for the pennymatching game. So let $i = \{\mathrm{ht}, \mathrm{tt}, \mathrm{th}\}$ and $a = \{\mathrm{h}, \mathrm{t}\}$. To compute $\mathsf{Uobs}(h)$ we first compute the forcing set $\delta(i) \cap a = \{\mathrm{hh}, \mathrm{ht}, \mathrm{tt}, \mathrm{th}\}$. We then compute the set of unexplained observations, we have only one maximal information set $i'' = \{\mathrm{ht}, \mathrm{tt}, \mathrm{th}\} \in \widehat{h}$, we subtract i'' from the forcing set and obtain $\{\mathrm{hh}\}$. For this set we compute the set of corresponding observations $\gamma(\{\mathrm{hh}\}) = \{\mathrm{h}/\mathbf{x}\}$. Since there is only one maximal information set, in this case, the intersection is trivially done: for i we get simply $O^\dagger = \{\mathrm{h}/\mathbf{x}\}$, and hence $\mathsf{Uobs}(h) = \{\langle\{\mathrm{ht}, \mathrm{tt}, \mathrm{th}\}, \{\mathrm{h}/\mathbf{x}\}\rangle\}$. $\triangleleft$

After explaining how to detect counter examples as unexplained observations, we now explain how to treat such a counterexample (i, o). First, let $h \Downarrow i$ be the *substrategy of h on i*, defined as $h \Downarrow i = \{\langle i', a' \rangle \in h \mid i' \subseteq i\}$. This represents the behaviour of the strategy $[\![h]\!]$ on all the information sets that are stronger-than-or-equal-to i. Now, if i has an unexplained observation $c^{\mathrm{out}}/c^{\mathrm{in}} \in [\![\mathsf{Uobs}(h)]\!](i)$, since all *stronger* $i' \subseteq i$ have a *weaker* allow set $h(i') \supseteq h(i)$ these stronger i' will also allow c^{out}. Hence, to effectively "treat" the unexplained observation CEDAR will replace the entire affected substrategy by the most permissive substrategy that is observation-closed on $o = c^{\mathrm{out}}/c^{\mathrm{in}}$. This most permissive substrategy will actually be the join of two substrategies, which are based on the restricted successor, and the controllable predecessor.

The controllable predecessor substrategy, illustrated in Figure 3 (a), contains the weakest, obstinate info/allow pairs that explain the observation by strengthening the information set to the weakest controllable region that forces the successor information set within one of the existing maximal information sets $i'' \in \widehat{h}$, i.e.: this new substrategy solves the problem by requiring *more knowledge*.

Definition 11 (Controllable Predecessor). For some strategy $h \in \llbracket \mathcal{C} \rrbracket [I, A]$, a set of maximal information sets $q \subseteq I$, and some observation $o = c^{\mathrm{out}}/c^{\mathrm{in}} \in O$ we let $\mathsf{Cpre}(h, q, o) \in \llbracket \mathcal{C} \rrbracket [I, A]$ be the *controllable $c^{\mathrm{out}}/c^{\mathrm{in}}$-predecessor strategy of h*, defined as follows:

$$\mathsf{Cpre}(h, q, o) = \llbracket \{ \langle i^c, a \rangle \mid \langle i, a \rangle \in h, i'' \in q, i^c = i \setminus \delta^{-1}((\delta(i) \cap o) \setminus i'') \} \rrbracket \qquad \lhd$$

The restricted successor substrategy, illustrated in Figure 3 (b), contains the weakest, obstinate info/allow pairs that avoid the unexplained observation by restricting the allow set and hence preventing the observation from arising at all, i.e.: this new substrategy solves the problem by becoming *less permissive*.

Definition 12 (Restricted Successor). We let $\mathsf{Rsucc}(h, c^{\mathrm{out}}) \in \llbracket \mathcal{C} \rrbracket [I, A]$ be the *restricted c^{out}-successor strategy of h*, defined as follows:

$$\mathsf{Rsucc}(h, c^{\mathrm{out}}) = \llbracket \{ \langle i^r, a^r \rangle \mid \langle i, a \rangle \in h, a^r = a \setminus \{c^{\mathrm{out}}\}, i^r = i \cap \delta^{-1}(\delta(i) \cap a^r) \} \rrbracket \lhd$$

Description of the Algorithm. We have now all prerequisites to present Algorithm 1, and illustrate its working on the blind penny matching example.

In line 1 the contravariant antichain is initialized to be fully uninformed, except that the system is not initially in a deadlock state, and maximally permissive, i.e. it allows *all* control outputs. The while condition in line 2 states what is basically the negation of observation-closedness, in terms of Definition 10. In line 3 we select a counterexample information set and concrete observation from the (symbolic) set of its unexplained observations. In line 4 we compute the most conservative refinement needed to make the contravariant antichain observation-closed for the selected counterexample observation. The most permissive substrategy is computed based on Definitions 11 and 12.

Note that this refinement is *strict* since (1) in the restricted successor, the allow set a^r of each newly introduced pair is guaranteed to be a strict subset of a, and (2) in the controllable predecessor, the information set i^c for each newly introduced pair is guaranteed to be a strict subset of i. Further note that, for (1), using a *saturated* contravariant antichain for info/allow pairs makes sure that besides being *strict*, the restricted successor

(a) controllable predecessors   (b) restricted successors
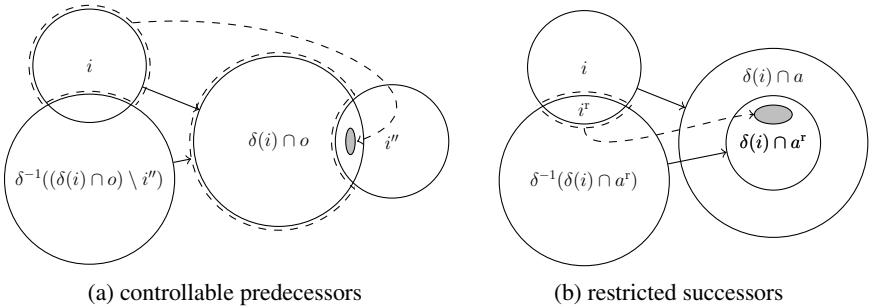
Fig. 3. Illustration of Definitions 11, 12: (a) for a given i, o and i'' we compute the weakest $i^c \subseteq i$ such that $\delta(i^c) \cap o \subseteq i''$, i.e.: i'' *explains* (i^c, o). Here, i^c is the left dashed region, and the gray region denotes $\delta(i^c) \cap o$. (b) for a given i, a and $a^r \subset a$ we compute the weakest $i^r \subseteq i$ such that $\langle i^r, a^r \rangle$ is obstinate. The gray region denotes $\delta(i^r) \cap a^r$.

strategy is also the *most permissive*, and, for (2), using a *sparse* contravariant antichain for the counterexample pairs makes sure that the new i^c pairs are never fully absorbed by the existing pairs above i in the saturated strategy.

After one iteration of the while loop we have obtained the next contravariant antichain which is strictly below the previous one but always above-or-equal-to f_G in the strategy subsumption ordering $\sqsupseteq$. The algorithm terminates when there are no more counterexamples to observation-closedness. Since the other requirement on f_G, obstinacy, is an invariant of the algorithm it follows that, at termination, $[\![h]\!] = f_G$. In line 5 we do a final test to see if f_G is winning from the initial state, if so then the results are useful and h is returned. If we are not interested in any particular initial state we may set $i^{\text{init}} = \varnothing$ since, upon termination, it always holds: $h(\varnothing) = f_G(\varnothing) = C^{\text{out}}$.

Algorithm 1. CEDAR (Counter Example Driven Antichain Refinement)

Data: $G = (L, C^{\text{out}}, C^{\text{in}}, \alpha, \beta, \delta, i^{\text{init}})$ — a game.
Result: the contravariant antichain h such that $[\![h]\!] = f_G$, or $\varnothing$ in case G is unsolvable.

1 $h \leftarrow \{ \langle \{\ell \in L \mid \delta(\ell) \neq \varnothing\}, C^{\text{out}} \rangle \}$
2 **while** $\mathsf{Uobs}(h) \neq \varnothing$ **do**
3 $\quad$ select some $\langle i, O^\dagger \rangle \in \mathsf{Uobs}(h)$ **and** $c^{\text{out}}/c^{\text{in}} \in O^\dagger$
4 $\quad$ $h \leftarrow (h \setminus h{\Downarrow}i) \sqcup (\mathsf{Rsucc}(h{\Downarrow}i, c^{\text{out}}) \sqcup \mathsf{Cpre}(h{\Downarrow}i, \widehat{h}, c^{\text{out}}/c^{\text{in}}))$
5 **if** $h(i^{\text{init}}) \neq \varnothing$ **then**
6 $\quad$ **return** h
7 **else**
8 $\quad$ **return** $\varnothing$

Example 5 (Solving pennymatching with CEDAR). The next table shows the successive values of the main program variables during a run of the algorithm on the blind pennymatching game of Example 1.

line **1**; initialization:	$h \leftarrow \{\langle\{\text{ht}, \text{th}, \text{tt}\}, \{\text{h}, \text{t}\}\rangle\}$
line **2**; observation closed?	$\mathsf{Uobs}(h) = \{\langle\{\text{ht}, \text{th}, \text{tt}\}, \{\text{h}/\mathbf{x}\}\rangle\}$
line **3**; select counterexample:	$i \leftarrow \{\text{ht}, \text{th}, \text{tt}\}$
	$c^{\text{out}}/c^{\text{in}} \leftarrow \text{h}/\mathbf{x}$
line **4**; refinement:	$\mathsf{Rsucc}(h{\Downarrow}i, \text{h}) = \{\langle\{\text{ht}, \text{th}, \text{tt}\}, \{\text{t}\}\rangle\}$
	$\mathsf{Cpre}(h{\Downarrow}i, \widehat{h}, \text{h}/\mathbf{x}) = \{\langle\{\text{th}\}, \{\text{h}, \text{t}\}\rangle\}$
	$h \leftarrow \{\langle\{\text{ht}, \text{th}, \text{tt}\}, \{\text{t}\}\rangle, \langle\{\text{th}\}, \{\text{h}, \text{t}\}\rangle\}$
line **2**; observation closed?	$\mathsf{Uobs}(h) = \varnothing$
line **5**; strategy is winning?	$h(\{\text{ht}\}) = \{\text{t}\}$
line **6**; yes, return h	$h = \{\langle\{\text{ht}, \text{th}, \text{tt}\}, \{\text{t}\}\rangle, \langle\{\text{th}\}, \{\text{h}, \text{t}\}\rangle\}$

As can be seen, on this simple example, the fixed point is reached after a single iteration. And the resulting strategy is indeed the *weakest* (and in this case *winning*) strategy. ◁

5   Experiments and Conclusion

We made a prototype implementation of the algorithm using the BuDDy package [12] for BDD manipulations. For comparison, we also implemented an on-the-fly, forward fixed point evaluation over the knowledge based subset construction as described by [3]. We refer to that algorithm as OTFOE (On-The-Fly fOrward Exploration). The comparison is not completely fair, because OTFOE computes a partial weakest strategy only for *reachable* information sets, and, vice versa, CEDAR does not compute the reachable information sets. Although the algorithms compute slightly different results, still it is interesting to contrast the fully symbolic approach with the explicit forward exploration.

We evaluated both algorithms on two different architectures. The first architecture is a single *monolithic* game graph. The second architecture is a *composed* game graph generated by four randomly synchronizing components of which only the first one contains a deadlock and control in- and outputs. Both architectures are illustrated in Figure 4. The distinguishing difference of the compositional architecture, as opposed to the monolithic architecture, lies in the concurrency and locality exhibited by the former and not by the latter.

The game components are randomly generated with a fixed number of locations and labels (cf. Figure 4). Random δ relations are generated componentwise. We take care that each δ_x is input enabled so that the product δ is always deadlock free. We then introduce deadlocks on a random set of error locations. For the transition density, $r_\delta = |\delta|/|L|^2$, we maintain $r_\delta = 0.3$ for the monolithic architecture and $0.04 < r_\delta < 0.08$ for the compositional architecture. Around these values we observed the maximal number of safely reachable information sets on average. Note that this biases our experiments to dense, solveable game graphs.

We solved 9 random games for each architecture and measured the number of $\delta(\cdot)$ and $\delta^{-1}(\cdot)$ operations performed. These are principal operations for both CEDAR and OTFOE. In particular, OTFOE uses one $\delta(\cdot)$ for determining the forcing set of each newly generated information set, and one $\delta^{-1}(\cdot)$ to test obstinacy whenever an allow set changes (i.e. for new states and for backpropagating unsafe control outputs).

The results of the experiments are shown in Table 1 and Figure 5. We see that, on dense game graphs generated by many components, CEDAR performs better than OTFOE. We still see evidence of bad worst-case behaviour in the form of outliers like random compositional game number 7. For sparser game graphs we observed the worst performance of CEDAR around $r_\delta = 0.1$ for monolithic games ($\approx 3.3 \cdot 10^3$ operations on average), and around $r_\delta = 0.01$ for compositional games (≈ 46 ops.). Whether
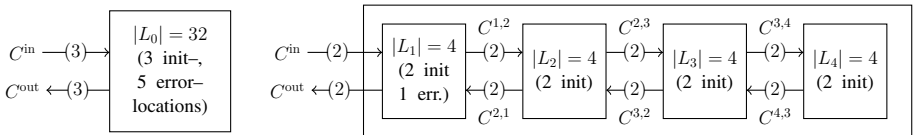
Fig. 4. The *monolithic* (left) and *compositional* (right) architectures. For the compositional architecture we make sure that component x is input enabled with respect to the internal synchronization labels in $C^{y,x}$. These special labels are projected away in the final game graph.

Table 1. Results for CEDAR and OTFOE on monolithic and compositional games. Under *ops.* we give the number of $\delta(\cdot)$ and $\delta^{-1}(\cdot)$ operations performed. Under *size* we give the *final* (*maximal*) size of the contravariant antichain for CEDAR and the *explored* (*safe*) information sets for OTFOE.

	monolithic games					compositional games			
	CEDAR		OTFOE			CEDAR		OTFOE	
#	ops.	size	ops.	size	#	ops.	size	ops.	size
1	18	4(4)	327	105(69)	1	13	2(2)	23	8(0)
2	207	1(8)	11	4(0)	2	9	3(3)	116	43(30)
3	290	1(12)	265	69(0)	3	16	3(3)	200	68(24)
4	90	6(8)	340	103(62)	4	16	4(4)	17	7(4)
5	62	5(6)	303	98(60)	5	42	1(4)	260	81(0)
6	37	5(5)	229	78(42)	6	9	3(3)	219	80(60)
7	203	1(11)	11	4(0)	7	77	3(4)	27	10(0)
8	314	5(14)	20	7(0)	8	15	3(3)	13	5(3)
9	212	1(10)	183	44(0)	9	27	4(5)	39	15(9)
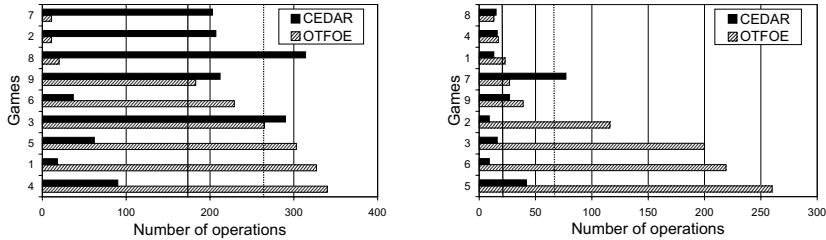
Fig. 5. Sorted charts of the data in Table 1; for *monolithic* (left) and *compositional* games (right). The dotted and solid lines show averages over 100 games for OTFOE and CEDAR, respectively.

or not bad worst-case behaviour plays a significant role on real instances needs to be evaluated by testing the algorithm on real-world models.

As future work, we suggest to investigate the degrees of freedom allowed in CEDAR, for instance in selecting the next counter example. Dynamic programming techniques could speed up the implementation, by avoiding the complete recomputation of the next counter example. In order to preserve memory usage, one could also store the strategies in a *sparse* contravariant antichain, and recompute its saturated pairs in each iteration. Finally, one could compare the efficiency of CEDAR with the antichain method in [2]. However, a fair comparison is complicated because both algorithms compute essentially different objects. In a separate paper, we will show how the results can be used for compositional controller synthesis.

References

1. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. Theoretical informatics and applications (July 2008)
2. Berwanger, D., Chatterjee, K., Doyen, L., Henzinger, T., Raje, S.: Strategy Construction for Parity Games with Imperfect Information. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 325–339. Springer, Heidelberg (2008)

3. Cassez, F.: Efficient On-the-Fly Algorithms for Partially Observable Timed Games. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 5–24. Springer, Heidelberg (2007)

4. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games with imperfect information. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)

5. de Alfaro, L., Henzinger, T.A.: Interface automata. In: FSE, pp. 109–120. ACM Press, New York (2001)

6. 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)

7. Liu, X., Ramakrishnan, C.R., Smolka, S.A.: Fully local and efficient evaluation of alternating fixed points (Extended abstract). In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, p. 5. Springer, Heidelberg (1998)

8. Mazala, R.: Infinite games. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 23–38. Springer, Heidelberg (2002)

9. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)

10. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. 25, 206–230 (1987)

11. Reif, J.H.: The complexity of two-player games of incomplete information. Journal of Computer and System Sciences 29, 274–301 (1984)

12. Lind-Nielsen, J.: Buddy: Binary decision diagrams,
    `http://sourceforge.net/projects/buddy`

13. Tirole, J., Fudenberg, D.: Game Theory. MIT Press, Cambridge (1991)

14. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)

15. De Wulf, M., Doyen, L., Raskin, J.-F.: A lattice theory for solving games of imperfect information. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 153–168. Springer, Heidelberg (2006)

Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads

Mohamed Faouzi Atig[1], Ahmed Bouajjani[1], and Shaz Qadeer[2]

[1] LIAFA, CNRS and University Paris Diderot, France
{atig,abou}@liafa.jussieu.fr
[2] Microsoft Research, Redmond
qadeer@microsoft.com

Abstract. Context-bounded analysis has been shown to be both efficient and effective at finding bugs in concurrent programs. According to its original definition, context-bounded analysis explores all behaviors of a concurrent program up to some fixed number of context switches between threads. This definition is inadequate for programs that create threads dynamically because bounding the number of context switches in a computation also bounds the number of threads involved in the computation. In this paper, we propose a more general definition of context-bounded analysis useful for programs with dynamic thread creation. The idea is to bound the number of context switches for each thread instead of bounding the number of switches of all threads. We consider several variants based on this new definition, and we establish decidability and complexity results for the analysis induced by them.

1 Introduction

The verification of multithreaded programs is a challenging problem both from the theoretical and the practical point of view. (We consider here programs with parallel threads which may use local variables as well as shared (global) variables.) Assuming that the variables of the program range over a finite domain (which can be obtained using some abstraction on the manipulated data), there are several aspects in multithreaded programs which make their analysis complex or even undecidable in general [13].

Indeed, it is well known that for instance in the case where each thread can be modeled as a finite-state system, the state space of the program grows exponentially w.r.t. the number of threads, and the reachability problem is PSPACE-hard. Moreover, if threads are modeled as pushdown systems, which corresponds to allowing unbounded depth (recursive) procedure calls in the program, then the reachability problem becomes undecidable as soon as two threads are considered.

Context-bounding has been proposed in [10] as a suitable technique for the analysis of multithreaded programs. The idea is to consider only the computations of the program that perform at most some fixed number of context switches between threads. (At each point only one thread is active and can modify the global variables, and a context-switch happens when the active thread terminates or is interrupted, and a pending one is activated.) The state space which must be explored may still be unbounded in presence of recursive procedure calls, but the context-bounded reachability problem

is decidable even in this case. In fact, context-bounding provides a very useful tradeoff between computational complexity and verification coverage. This tradeoff is based on three important properties. First, context-bounded verification can be performed more efficiently than unbounded verification. From the complexity-theoretic point of view, it can be seen that context-bounded reachability is an NP-complete problem (even in the case of pushdown threads). Second, many concurrency errors, such as data races and atomicity violations, are manifested in executions with few context switches [9]. Finally, verifying all executions of a concurrent program up to a context bound provides an intuitive and meaningful notion of coverage to the programmer.

In the last few years, several implementations and algorithmic improvements have been proposed for context-bounded verification [2,9,16,7,6]. For instance, context-bounded verification has been implemented in explicit-state model checkers such as CHESS [9] and SPIN [18]; it has also been implemented in symbolic model checkers such as SLAM [11], jMoped [16], and in [6].

While the concept of context-bounding is adequate for multithreaded programs with a (fixed) finite number of threads, the question we consider in this paper is whether this concept is still adequate when dynamic creation of threads is considered.

Dynamic thread creation is useful for modeling several important aspects, e.g., (1) unbounded number of concurrently execution of software modules such as file systems, device drivers, non-blocking data structures etc., or (2) creation of asynchronous activity such as forking a thread, queuing a closure to a threadpool with or without timers, callbacks, etc. Both these sources are very important for modeling operating system components; they are likely to become important even for application software as it becomes increasingly parallel in order to harness the power of multi-core architectures.

We argue that the "classical" notion of context-bounding which has been used so far in the existing work is actually too restrictive in this case. Indeed, bounding the number of context switches in a computation also bounds the number of threads involved. In this paper, we propose a more general definition of context-bounded analysis useful for programs with dynamic thread creation. The idea is to bound the number of context switches for each thread instead of bounding the number of switches of all threads. We consider several variants based on this new definition, and we establish decidability and complexity results for the analysis induced by them.

We introduce a notion of K-bounded computations where each of the involved threads can be interrupted and resumed at most K times. (We consider that when a thread is created, the number of context switches it can perform is the one of its ancestor minus 1.) Notice that the number of context switches by all threads in a computation is not bounded since the number of threads involved is not bounded.

In the case of finite-state threads, we prove that this problem is as hard as the coverability problem for Petri nets (which is EXPSPACE-complete). The reduction from our problem to the coverability problem of Petri nets is based on the simple idea of counting the number of pending threads for different values of the global and local states, as well as of the number of switches that these threads are allowed to perform. conversely, we prove that the coverability problem of Petri nets can be reduced to the 2-bounded reachability problem. These results show that in the case of dynamic thread creation, considering the notion of context-bounding for each individual thread makes the

complexity jumps from NP-completeness to EXPSPACE-completeness, even in the case of finite-state threads. Then, an interesting question is whether it is possible to have a notion of context-bounding with a lower complexity. We propose for that the notion of stratified context-bounding. The idea is to consider computations where the scheduling of the threads is ordered according to their number of allowed switches: First, threads of level K (the level means here the number of allowed switches) are scheduled generating threads of level $K-1$, then threads of level $K-1$ are scheduled, and so on. Next, it is possible to schedule again threads of level K and repeat the process, but this only for a finite number of times L. Again, notice that (K,L)-stratified computations may have an unbounded number of context switches since it is possible to schedule an unbounded number of threads at each level. This concept generalizes obviously the "classical" notion of context-bounding. We prove that, for finite-state threads, the (K,L)-stratified context-bounded reachability problem is NP-complete (i.e., it matches the complexity of the "classical" context-bounded reachability problem). The proof is by a reduction to the satisfiability problem of existential Presburger formulas.

Then, we consider the case of dynamic creation of pushdown threads. We prove that, surprisingly, the K-bounded reachability problem is in fact decidable, and that the same holds also for the (K,L)-stratified context-bounded reachability problem. To establish these results, we prove that these problems (for pushdown threads) can be reduced to their corresponding problems for finite-state threads. This reduction is not trivial. The main ideas behind the reduction are as follows: First, the K-bounded behaviors of each single thread can be represented by a labeled pushdown system which (1) makes visible (as labels) on its transitions the created threads, and (2) guesses points of interruption-resumption and the corresponding values of the global states. (These guesses are also made visible on the transitions.) Then, the main problem is to "synchronize" these labeled pushdown systems so that their guesses can be validated. The key observation is that it is possible to abstract these systems without loss of preciseness by finite-state systems. This is due to the fact that we can consider that some of the generated threads can be lost (since they can be seen as threads that are never activated), and therefore we can reason about the downward closure of the languages of the labeled pushdown systems mentionned above (w.r.t. suitable sub-word relation). This downward closure is in fact always regular and effectively constructible.

2   Preliminaries

Words and languages.   Let Σ be a finite alphabet. We denote by Σ^* (resp. Σ^+) the set of all *words* (resp. non empty words) over Σ, and by ε the empty word. A language L is a (possibly infinite) set of words. Let $u \in \Sigma^*$ and $a \in \Sigma$. We denote by $|u|$ the length of u and by $|u|_a$ the number of occurrences of a in u. Consider a non empty word $u = a_1 \cdots a_n$. For any i such that $1 \leq i \leq n$, we denote by u_i the symbol a_i.

Given an alphabet Σ, we denote by $\preceq \subseteq \Sigma^* \times \Sigma^*$ the *subword relation* defined as follows: for every $u = a_1 \cdots a_n \in \Sigma^*$, and every $v = b_1 \cdots b_m \in \Sigma^*$, $u \preceq v$ iff $\exists i_1, \ldots, i_n \in \{1, \ldots, m\}$ such that $i_1 < i_2 < \ldots < i_n$ and $\forall j \in \{1, \ldots, n\}, a_j = b_{i_j}$. Given a language $L \subseteq \Sigma^*$, the *downward closure* of L (w.r.t. $\preceq$) is the set $L \downarrow = \{u \in \Sigma^* \mid \exists v \in L, u \preceq v\}$.

Finite State Automata. A Finite State Automaton (FSA) is a tuple $S = (S, \Sigma, \delta, s^{init}, s^{final})$ where S is a finite set of states, Σ is a finite input alphabet, $\delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is a finite set of transitions, $s^{init} \in S$ is the initial state, and $s^{final} \in S$ is the acceptor state. The language accepted by the finite state automaton S is denoted $L(S)$.

Labeled Pushdown Systems. A Labeled Pushdown System (LPDS) is defined by a tuple $\mathcal{P} = (G, \Sigma, \Gamma, \Delta)$ where G is a finite set of states, Σ is an input alphabet (actions), Γ is a stack alphabet, and Δ is a finite set of transition rules of the form: $g\gamma \xrightarrow{a} g'w'$ where $g, g' \in G$, $a \in \Sigma \cup \{\varepsilon\}$, $\gamma \in \Gamma$, and $w' \in \Gamma^*$ such that $|w'| \leq 2$.

A *configuration* of $\mathcal{P}$ is a tuple $\langle g, \sigma, w \rangle$ where $g \in G$ is a state, $\sigma \in \Sigma^*$ is an input word, and $w \in \Gamma^*$ is a stack content. We define the binary relation $\Rightarrow_{\mathcal{P}}$ between configurations as follows: $\langle g, a\sigma, \gamma w \rangle \Rightarrow_{\mathcal{P}} \langle g', \sigma, w'w \rangle$ iff $g\gamma \xrightarrow{a} g'w' \in \Delta$. The transition relation $\Rightarrow_{\mathcal{P}}^*$ is the reflexive transitive closure of the binary relation $\Rightarrow_{\mathcal{P}}$.

Given a labeled pushdown system $\mathcal{P} = (G, \Sigma, \Gamma, \Delta)$, two states $g, g' \in G$, and a stack symbol γ, let $L_{\mathcal{P}}(g\gamma, g') = \{\sigma \in \Sigma^* \mid \exists w \in \Gamma^* \; s.t \; \langle g, \sigma, \gamma \rangle \Rightarrow_{\mathcal{P}}^* \langle g', \varepsilon, w \rangle\}$. Clearly, $L_{\mathcal{P}}(g\gamma, g')$ is a context-free language, and conversely, every context-free language can be defined as a trace language of some labeled pushdown system.

We recall hereafter a result due to Courcelle [3] which will be used later in the paper.

Theorem 1. *Let $\mathcal{P} = (G, \Sigma, \Gamma, \Delta)$ be a LPDS, $g, g' \in G$ be two states, and $\gamma \in \Gamma$ be a stack symbol. Then, it is possible to construct a FSA $S = (S, \Sigma, \delta, s^{init}, s^{final})$ such that $L(S) = (L_{\mathcal{P}}(g\gamma, g')) \downarrow$, where in the worst case $|S|$ is exponential in $(|G| + |\Sigma| + |\Gamma|)$.*

Multi-sets. Let Ξ be a nonempty alphabet (possibly infinite). A *multi-set* over Ξ is a function $M : \Xi \to \mathbb{N}$. We denote by $M[\Xi]$ the collection of all multi-sets over Ξ and by $\emptyset$ the empty multi-set. Given two multi-sets M and M', we write $M' \leq M$ iff $M'(a) \leq M(a)$ for every $a \in \Xi$; and $M + M'$ (resp. $M - M'$ if $M' \leq M$) to denote the multi-set where $(M + M')(a) = M(a) + M'(a)$ (resp. $(M - M')(a) = M(a) - M'(a)$) for every $a \in \Xi$. For every word $u \in \Xi^*$, $[u]$ is the multi-set such that $[u](a) = |u|_a$ for every $a \in \Xi$. Sometimes, $[u]$ is called the Parikh image of u. This definition is extended in the straightforward manner to languages (sets of words) as follows $[L] = \{[u] \; : \; u \in L\}$.

Petri Nets. A Petri net is a pair $\mathcal{N} = (P, T)$ where P is a finite set of places and $T \subseteq P^* \times P^*$ is a finite set of transition rules. We often write $w \rightarrowtail w'$ to denote a transition $(w, w') \in T$. Given a transition $t = w \rightarrowtail w' \in T$, we define a relation $\xrightarrow{t} \subseteq (M[P] \times M[P])$ between multi-sets over P as follows: $W \xrightarrow{t} W'$ iff $W \geq [w]$ and $W' = W + [w'] - [w]$. We define the transition relation $\rightarrow_{\mathcal{N}}$ on multi-sets over P by the union of $\xrightarrow{t}$, i.e., $\rightarrow_{\mathcal{N}} = \bigcup_{t \in T} \xrightarrow{t}$. The transition relation $\rightarrow_{\mathcal{N}}^*$ is the reflexive transitive closure of $\rightarrow_{\mathcal{N}}$.

The coverability problem for a Petri net $\mathcal{N}$ is the problem of deciding for two given places p and p' whether there is a multi-set W such that $[p'] \leq W$ and $[p] \rightarrow_{\mathcal{N}}^* W$.

Theorem 2. *The coverability problem for Petri nets is EXPSPACE-complete* [8,12].

Existential Presburger Formulas (EPF). Let $\mathcal{V}$ be a set of variables. We use $x, y, \dots$ to range over variables in $\mathcal{V}$. The set of *existential* Presburger formulas is defined by the following grammar and interpreted over natural numbers:

$$t ::= 0 \mid 1 \mid x \mid t_1 + t_2 \qquad \phi ::= t_1 > t_2 \mid t_1 = t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x \cdot \phi_1$$

The semantics of these formulas is defined in the standard way. Given a formula ϕ with free variables $x_1, \ldots, x_n$, and a valuation $U : \mathcal{V} \to \mathbb{N}$, we denote $\phi(U)$ the truth value of ϕ for the valuation U. We say that a formula ϕ is satisfiable if there is some valuation U such that $\phi(U)$ is true. It is well-know that the satisfiability problem for existential Presburger formulas is decidable [17], and that:

Theorem 3. *The satisfiability problem for existential Presburger formulas is NP-complete.*

Given a language L over the alphabet $\Sigma = \{a_1, \ldots, a_n\}$, the set $[L]$ (called the Parikh image of L) is definable by a formula ϕ with free variables $x_{a_1}, \ldots, x_{a_n}$ if for every valuation U, $\phi(U)$ is true iff there is a word $u \in L$ such that $U(x_{a_i}) = [u](a_i)$ for every $i \in \{1, \ldots, n\}$. We recall a result about existential Presburger formulas given in [14,17].

Theorem 4. *Let $\mathcal{P} = (G, \Sigma, \Gamma, \Delta)$ be a LPDS, $g, g' \in G$ be two states, and $\gamma \in \Gamma$ be a stack symbol. Then, it is possible to compute in polynomial time an existential Presburger formula ϕ which defines $[L_\mathcal{P}(g\gamma, g')]$.*

3   Dynamic Networks of Concurrent Systems

3.1   Syntax

A Dynamic Network of Concurrent Pushdown System (DCPS) is a tuple $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ where G is a finite set of *states*, Γ is a finite set of *stack symbols*, g_0 is the initial state, γ_0 is the initial stack symbol, and Δ is a finite sets of transition rules of the forms: (1) $g\gamma \hookrightarrow g'w'$, or (2) $g\gamma \hookrightarrow g'w' \rhd \gamma'$ where $g, g' \in G$, $\gamma, \gamma' \in \Gamma$, $w' \in \Gamma^*$, and $|w'| \leq 2$.

A DCPS models dynamic multithreaded programs with (potentially recursive) procedure calls. Threads are modeled as pushdown processes which can spawn new processes. They have local variables and have also access to global (shared) variables. The values of the local variables are modeled using the stack alphabet Γ, whereas the values of the global variables are modeled using states in G. Rules of the form $g\gamma \hookrightarrow g'w'$ correspond to standard transitions of pushdown systems (popping γ and then pushing w' while changing the control state from g to g'), and rules of the form $g\gamma \hookrightarrow g'w' \rhd \gamma'$ are similar but in addition they create a new thread with an initial local state γ'. Notice that push (resp. pop) operations allow to model procedure calls (resp. returns).

When unbounded recursion is not considered, threads can be modeled as finite-state processes instead of pushdown systems. This corresponds to the special case where in all the transition rules defined above the pushed sequence w' is of size at most 1. We use the acronym DCFS for dynamic networks of concurrent finite-state system.

3.2   Semantics

A configuration of $\mathcal{A}$ is given by (1) a state g (the current value of the global store), (2) the local configuration of the *active* thread, which is a pair (w, i) where w is its call stack and i is its switch number (the number of interruptions/resumptions of the thread together with the switch number of its ancestor at the moment of its creation), and

(3) a multiset of the local configurations of the idle threads. Formally, a configuration of $\mathcal{A}$ is a tuple $\langle g, (w,i), M \rangle \in G \times (\Gamma^* \times \mathbb{N}) \times M[\Gamma^* \times \mathbb{N}]$. We assume that the initial configuration of $\mathcal{A}$ is $(g_0, (\gamma_0, 0), \emptyset)$.

For a given $i \in \mathbb{N}$, the relation $\Rightarrow_i$ on configurations is $\rightarrow_i \cup \mapsto_i$, where $\rightarrow_i$ and $\mapsto_i$ are defined as follows:

- $\langle g, (\gamma w, i), M \rangle \rightarrow_i \langle g', (w'w, i), M' \rangle$ iff (1) there is a rule $g\gamma \hookrightarrow g'w' \in \Delta$ and $M = M'$, or (2) there is a rule $g\gamma \hookrightarrow g'w' \rhd \gamma' \in \Delta$ and $M' = M + [(\gamma', i+1)]$.
- $\langle g, (w,i), M + [(w', j)] \rangle \mapsto_i \langle g, (w', j), M + [(w, i+1)] \rangle$ for every $j \in \mathbb{N}$, $g \in G$, $M \in M[\Gamma^* \times \mathbb{N}]$, and $w, w' \in \Gamma^*$.

The relations $\rightarrow_i$ correspond to the execution of pushdown (pop and push) operations, with the possibility of creating new threads (added to the multiset of idle threads). The created threads get the switch number $i+1$. The relations $\mapsto_i$ correspond to context switches: The local configuration (w', j) of a waiting thread is taken from the multiset and given the status of active, while the local configuration (w, i) of the interrupted task is stored in the multiset after incrementing its switch number.

Let $\Rightarrow_{\leq B} = \bigcup_{i \leq B} \Rightarrow_i$ for every $B \in \mathbb{N} \cup \{\infty\}$. We write simply $\Rightarrow$ instead of $\Rightarrow_{\leq \infty}$. Finally, $\Rightarrow_i^*$ and $\Rightarrow_{\leq B}^*$ denote the transitive closure of $\Rightarrow_i$ and $\Rightarrow_{\leq B}$, respectively.

3.3   Reachability Problems

We consider the three following notions of reachability:

State reachability: A state g is said to be reachable iff $\langle g_0, (\gamma_0, 0), \emptyset \rangle \Rightarrow^* \langle g, (w, j), M \rangle$ for some $(w, j) \in \Gamma^* \times \mathbb{N}$ and $M \in M[(\Gamma^* \times \mathbb{N})]$. The state reachability problem (SRP for short) is, for a given DCPS $\mathcal{A}$ and $g \in G$, to determine whether g is reachable by $\mathcal{A}$.

K-bounded state reachability: Given $K \in \mathbb{N}$, a state $g \in G$ is said to be K-reachable iff $\langle g_0, (\gamma_0, 0), \emptyset \rangle \Rightarrow_{\leq K}^* \langle g, (w, j), M \rangle$ for some $(w, j) \in \Gamma^* \times \{0, \ldots, K\}$, and $M \in M[\Gamma^* \times \{0, \ldots, K+1\}]$. The K-bounded state reachability problem (SRP[K] for short) is, for a given DCPS $\mathcal{A}$, $K \in \mathbb{N}$, and $g \in G$, to determine whether g is K-reachable by $\mathcal{A}$.

Observe that, in SRP[K], a bound K is imposed on the number of switches (interruptions/resumptions) performed by each thread (together with the switch number of its ancestor at the moment of its creation). However, due to dynamic creation of threads, bounding the number of switches of each thread does not bound the number of switches in the whole computations of the system (since an arbitrarily large number of threads can be involved in these computations).

L-stratified K-bounded state reachability: Given $K, L \in \mathbb{N}$, a state $g \in G$ is said to be $[K, L]$-reachable iff $\langle g_0, (\gamma_0, 0), \emptyset \rangle (\Rightarrow_0^* \circ \cdots \circ \Rightarrow_K^*)^L \langle g, (w, j), M \rangle$ for some $(w, j) \in \Gamma^* \times \{0, \ldots, K\}$, and $M \in M[\Gamma^* \times \{0, \ldots, K+1\}]$. The L-stratified K-bounded state reachability problem (SRP[K, L] for short) is, for a given DCPS $\mathcal{A}$, $g \in G$, and $K, L \in \mathbb{N}$, to determine whether g is $[K, L]$-reachable.

In SRP[K, L], a special kind of K-bounded computations (called stratified computations) are considered: In one stratum of such a computation, threads are scheduled according to their increasing switch number (from 0 to K). This corresponds to the consideration of the relation $\Rightarrow_0^* \circ \cdots \circ \Rightarrow_K^*$. Then, a L-stratified computation is a sequence

of L strata. Observe that even in the case of stratified computations, an arbitrarily large number of context switches may occur along a computation due to dynamic creation of threads. Moreover, relaxing the bound L, i.e. considering arbitrarily large sequences of strata, corresponds to considering the SRP[K] problem. Very particular stratified computations are those where the whole number of context switches is bounded [10].

4 Analysis of Dynamic Networks of Concurrent Finite-State Systems

In this section, we show that SRP and SRP[K] are EXPSPACE-complete (Theorem 5), whereas, the problem SRP[K, L] is NP-complete (Theorem 6).

Theorem 5. *The problems SRP and SRP[K] for DCFSs, for every natural number $K \geq 2$, are EXPSPACE-complete.*

Proof: We prove that SRP[K] for DCFSs is polynomially reducible to the coverability problem for Petri nets and vice-versa. We give here a sketch of the proof (for more details see [1]). The constructions presented below can be adapted to show that the SRP problem for DCFSs is also EXPSPACE-complete.

From SRP[K] for DCFSs to coverability problem for Petri nets. Given a natural number K and a DCFS $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$, we construct a Petri net $\mathcal{N} = (P, T)$ which has the following structure:

- The set of places:
    - A place (w, j) is associated with each natural number $j \in \{0, \ldots, K+1\}$ and each stack configuration $w \in \Gamma \cup \{\varepsilon\}$. The number of tokens in the place (w, j) is the number of pending threads of $\mathcal{A}$ with local configuration (w, j).
    - A place (g, w, i) is associated with each $i \in \{0, \ldots, K\}$, each state $g \in G$, and each stack configuration $w \in \Gamma \cup \{\varepsilon\}$. A token in the place (g, w, i) represents the fact that g is the current value of the global store of $\mathcal{A}$ and that (w, i) is the local configuration of the active thread.
- The set of transitions:
    - For each $i \in \{0, \ldots, K\}$ and each rule $g_1 \gamma \hookrightarrow g_2 w$ (resp. $g_1 \gamma \hookrightarrow g_2 w \rhd \gamma'$) of $\mathcal{A}$, $\mathcal{N}$ has a transition $(g_1, \gamma, i) \rightarrowtail (g_2, w, i)$ (resp. $(g_1, \gamma, i) \rightarrowtail (g_2, w, i)(\gamma', i+1)$).
    - For each $i, j \in \{0, \ldots, K\}$, each state $g \in G$, and each pair of stack configurations $w, w' \in \Gamma \cup \{\varepsilon\}$, there is a transition $(g, w, i)(w', j) \rightarrowtail (g, w', j)(w, i+1)$ in T. This transition simulates a context switch of $\mathcal{A}$.

Notice that the size of $\mathcal{N}$ is polynomial in the size of $\mathcal{A}$.

Lemma 1. $(g_0, (\gamma_0, 0), \emptyset) \Rightarrow^* (g, (w, i), M)$ *for some* $(w, i) \in (\Gamma \cup \{\varepsilon\}) \times \{0, \ldots, K\}$ *and* $M \in M[(\Gamma \cup \{\varepsilon\}) \times \{0, \ldots, K+1\}]$ *iff* $[(g_0, \gamma_0, 0)] \rightarrow^*_{\mathcal{N}} [(g, w, i)] + M$.

From coverability problem for Petri nets to SRP[2] for DCFSs. Given a Petri net $\mathcal{N} = (P, T)$ and two places $p_0, p_f \in P$, we construct a DCFS $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ such that: the state g_f is 2-reachable by $\mathcal{A}$ iff there is a multi-set $W \in M[P]$ such that $[p_0] \rightarrow^*_{\mathcal{P}} W$ and $[p_f] \leq W$. Intuitively, $\mathcal{A}$ has a special stack symbol γ_1 such that the number of pending threads with local configuration $(\gamma_1, 1)$ gives an upper bound of the length

of the run of $\mathcal{N}$ simulated by $\mathcal{A}$. For each place $p \in P$, $\mathcal{A}$ has a stack symbol p; the number of such pending threads with stack content p denotes the current number of tokens in p. We now sketch the behavior of $\mathcal{A}$. The DCFS $\mathcal{A}$ guesses the length of the simulated run of $\mathcal{N}$ by creating a number of threads with local configuration $(\gamma_1, 1)$ from the initial configuration. Then, the simulation of a rule $t = w \rightarrowtail w' \in T$ is done in two steps. First, $\mathcal{A}$ checks if t can be fired by verifying if there is a pending thread with local configuration $(w_i, 2)$ for every $i \in \{1, \ldots, |w|\}$. Second, $\mathcal{A}$ uses pending threads with local configuration $(\gamma_1, 1)$ to create, for every $j \in \{1, \ldots, |w'|\}$, a thread with local configuration $(w'_j, 2)$. Finally, to check if there is a token in the place p_f, $\mathcal{A}$ verifies if there a pending thread with local configuration $(p_f, 2)$ and moves its state to g_f. Formally, $\mathcal{A}$ is built up from $\mathcal{N}$ as follows:

- The set of states:
  - $\mathcal{A}$ has two special states g_0 and g_f. The states g_0 and g_f are the initial state and the final state, respectively.
  - For each rule $t = w \rightarrowtail w' \in T$, $\mathcal{A}$ has the following sequences of global states $g_{(t,w_1)}, \ldots, g_{(t,w_{|w|})}$ and $g_{(t,w'_1)}, \ldots, g_{(t,w'_{|w'|})}$. The sequence of states $g_{(t,w_1)}, \ldots, g_{(t,w_{|w|})}$ is used to simulate taking iteratively a token from each place w_i for $i = 1$ to $|w|$. The sequence of states $g_{(t,w'_1)}, \ldots, g_{(t,w'_{|w'|})}$ is used to simulate adding iteratively a token to each place w'_j for $j = 1$ to $|w'|$.
- The set of stack alphabet:
  - $\mathcal{A}$ has two special stack symbols γ_0 and γ_1. The symbol γ_0 represents the initial stack content. Symbols γ_1 represent auxiliary threads that are "*consumed*" for simulating tokens generation by transitions of $\mathcal{N}$.
  - For each place $p \in P$, $\mathcal{A}$ has a stack symbol p. The number of pending threads with stack content p denotes the current number of token in the place p.
- The set Δ is the smallest set of rules satisfying the following conditions:
  - **Guessing the length of the run of $\mathcal{N}$:** The rule $g_0\gamma_0 \hookrightarrow g_0\gamma_0 \triangleright \gamma_1$ is in Δ. This rule creates an arbitrary number of threads γ_1 with switch number 1. This gives an upper bound of the length of the run of $\mathcal{N}$ simulated by $\mathcal{A}$.
  - **Creation of the initial marking of $\mathcal{N}$:** The rule $g_0\gamma_0 \hookrightarrow g_0 \triangleright p_0$ is in Δ. This rule creates a thread p_0. This corresponds to the initial multiset $[p_0]$ of $\mathcal{N}$.
  - **Simulation of a transition rule of $\mathcal{N}$:** A transition $t = w \rightarrowtail w'$ is simulated by checking iteratively that there is a pending thread with stack content w_i for $i = 1$ to $|w|$, and then, by creating iteratively a thread with initial stack content w'_j for $j = 1$ to $|w'|$.
    * **Initialization of the simulation:** For each transition $t = w \rightarrowtail w' \in T$, the rule $g_0 w_1 \hookrightarrow g_{(t,w_1)}$ is in Δ. This rule corresponds to start simulating t and to check that there is a pending thread with stack content w_1.
    * **Checking if the transition rule can be fired:** For each transition $t = w \rightarrowtail w' \in T$ and for each $i \in \{1, \ldots, |w| - 1\}$, the rules $g_{(t,w_i)} w_{i+1} \hookrightarrow g_{(t,w_{i+1})}$ are in Δ. These transitions simulate the operation of taking a token from each place in $w_2, \ldots, w_{|w|}$.
    * **Generating the output tokens:** For each transition $t = w \rightarrowtail w' \in T$ and for each $j \in \{1, \ldots, |w'| - 1\}$, the rules $g_{(t,w_{|w|})}\gamma_1 \hookrightarrow g_{(t,w'_1)}\gamma_1 \triangleright w'_1$,

$g_{(t,w'_j)}\gamma_1 \hookrightarrow g_{(t,w'_{j+1})}\gamma_1 \rhd w'_{(t,j+1)}$, and $g_{(t,w'_{|w'|})}\gamma_1 \rightarrowtail g_0$ are in Δ. These transitions simulate the operation of adding a token to each place of $w'_1,\ldots,w'_{|w'|}$. Notice that if the thread γ_1 has a switch number 1, then the created threads $w'_1,\ldots,w'_{|w'|}$ have switch number 2.

- **Checking the final marking**: The rule $g_0 p_f \hookrightarrow g_f$ is in Δ. This corresponds to checking if there a multiset with at least one token in p_f reachable by $\mathcal{N}$.

The relation between $\mathcal{N}$ and $\mathcal{A}$ is given by the following lemma:

Lemma 2. *The control state g_f is 2-reachable by $\mathcal{A}$ iff there a marking $W \in M[P]$ such that $W \geq [p_f]$ and $[p_0] \rightarrow^*_{\mathcal{N}} W$.* □

We consider now the problem $\mathrm{SRP}[K,L]$ for $K,L \in \mathbb{N}$. We prove that:

Theorem 6. *For every $K,L \in \mathbb{N}$, the problem $\mathrm{SRP}[K,L]$ for DCFSs is NP-complete.*

Proof: NP-hardness is proved by a reduction from the coverability problem for acyclic Petri nets [15] to $\mathrm{SRP}[K,1]$. This is done by a simple adaptation of the construction given in Theorem 5. The upper-bound is obtained by a reduction to the satisfiability problem of *existential* Presburger formulas. We sketch hereafter the proof for the special when $L = 1$. The extension to $L > 1$ is straightforward [1].

Let $\mathcal{A} = (G,\Gamma,\Delta,g_0,\gamma_0)$ be a DCFS and K be a natural number. We recall that a state $g_{K+1} \in G$ is $[K,1]$-reachable by $\mathcal{A}$ iff there is a sequence of states $g_1,\ldots,g_K \in G$, a sequence of stack configurations $w_1,\ldots,w_{K+1} \in \Gamma \cup \{\varepsilon\}$, and a sequence of multi-sets $M_1,\ldots,M_{K+1} \in M[(\Gamma \cup \{\varepsilon\}) \times \{0,\ldots,K+1\}]$ such that:

$$\langle g_0,(\gamma_0,0),\emptyset \rangle \Rightarrow^*_0 \langle g_1,(w_1,1),M_1 \rangle \Rightarrow^*_1 \cdots \Rightarrow^*_K \langle g_{K+1},(w_{K+1},K+1),M_{K+1} \rangle \quad (a)$$

The problem of checking whether a state g_{K+1} is $[K,1]$-reachable by $\mathcal{A}$ can be rewriting as follows: Whether there are some $w_0,\ldots,w_{K+1} \in \Gamma \cup \{\varepsilon\}$, $g_1,\ldots,g_K \in G$, and $N_j,N'_j \in M[\Gamma \times \{j\}]$ for every $j \in \{0,\ldots,K+1\}$, such that: (1) $w_0 = \gamma_0$, (2) $N_0 = N'_0 = \emptyset$, (3) and for every $i \in \{0,\ldots,K\}$, we have:

$$\langle g_i,(w_i,i),N'_i + \sum_{l=0}^{i-1}(N'_l - N_l) \rangle \Rightarrow^*_i \langle g_{i+1},(w_{i+1},i+1),N'_{i+1} + \sum_{l=0}^{i}(N'_l - N_l) \rangle \quad (b)$$

Observe that for every $i \in \{0,\ldots,K+1\}$, we have that $M_i = N'_i + \sum_{l=0}^{i-1}(N'_l - N_l)$ and that $N_i + [(w_i,i)]$ is the multi-set of executed threads with switch number i. Then, the equation (b) can be rewritten as follows: For every $i \in \{0,\ldots,K\}$, we have:

$$\langle g_i,(w_i,i),N_i \rangle \Rightarrow^*_i \langle g_{i+1},(w_{i+1},i+1),N'_{i+1} \rangle \text{ and } N_i \leq N'_i \quad (c)$$

This means that $\mathcal{A}$ can reach the configuration $\langle g_{i+1},(w_{i+1},i+1),N'_{i+1} \rangle$ while executing all threads in the multiset $[(w_i,i)] + N_i$ which should be less than the number of generated threads with switch number i, i.e. $[(w_i,i)] + N'_i$. We can further simplify our problem as follows: A state g_{K+1} is $[K,1]$-reachable by $\mathcal{A}$ iff there are some $w_0,w'_0,w_1,\ldots,w_{K+1},w'_{K+1} \in \Gamma \cup \{\varepsilon\}$, $g_1,\ldots,g_K \in G$, and $N_j,N''_j \in M[\Gamma \times \{j\}]$ for every $j \in \{0,\ldots,K+1\}$, such that: (1) $w_0 = w'_0 = \gamma_0$, (2) $N_0 = N''_0 = \emptyset$, (3) and for every $i \in \{0,\ldots,K\}$, we have:

$$\langle g_i,(w_i,i),N_i \rangle \Rightarrow^*_i \langle g_{i+1},(w'_{i+1},i+1),N''_{i+1} \rangle \text{ and } N_i + [(w_i,i)] \leq N''_i + [(w'_i,i)] \quad (d)$$

Observe that if (c) holds, then (d) holds by simply considering $w_i' = w_i$ for every $i \in \{0, \ldots, K+1\}$. On the other hand, if (d) is true, then (c) is true by taking $N_{K+1}' = N_{K+1}''$, $w_{K+1} = w_{K+1}'$, and for every $i \in \{0, \ldots, K\}$, $N_i' = N_i'' + [(w_i', i)] - [(w_i, i)]$ which possible since $N_i + [(w_i, i)] \leq N_i'' + [(w_i', i)]$. In the latter case, we can show that for every $i \in \{0, \ldots, K\}$, $\langle g_i, (w_i, i), N_i \rangle \Rightarrow_i^* \langle g_{i+1}, (w_{i+1}, i+1), N_{i+1}' \rangle$ and $N_i \leq N_i'$.

Hence, a state g_{K+1} is $[K, 1]$-reachable iff the three following requirements hold:

1. For each $i \in \{0, \ldots, K\}$, we have $\langle g_i, (w_i, i), N_i \rangle \Rightarrow_i^* \langle g_{i+1}, (w_{i+1}', i+1), N_{i+1}'' \rangle$.
2. For each $i \in \{1, \ldots, K\}$, we have $N_i + [(w_i, i)] \leq N_i'' + [(w_i', i)]$.
3. $N_0 = N_0'' = \emptyset$ and $w_0 = w_0' = \gamma_0$.

(Notice indeed that requirements 1, 2 and 3 are equivalent to (d)).

Let us fix (guess) a sequence of states $\sigma = g_1, \ldots, g_K \in G$. We show how to compute an existential Presburger formula ϕ_σ, of polynomial size in the size of $\mathcal{A}$, which is satisfiable iff the state g_{K+1} is $[K, 1]$-reachable by $\mathcal{A}$. To this end, we compute an existential Presburger sub-formula for each of the previous three requirements. These sub-formulas use the set $\mathcal{V} = \{x_{(w,i)}, y_{(w,i)} \mid w \in (\Gamma \cup \{\varepsilon\}) \wedge 0 \leq i \leq K+1\}$ as a set of free variables such that for each $i \in \{0, \ldots, K+1\}$ and for each $w \in \Gamma \cup \{\varepsilon\}$, the variable $x_{(w,i)}$ (resp. $y_{(w,i)}$) stands for the number of occurrences of (w, i) in the multiset $([(w_i, i)] + N_i)$ (resp. $([(w_i', i)] + N_i'')$), i.e. $([(w_i, i)] + N_i)((w, i))$ (resp. $([(w_i', i)] + N_i'')((w, i))$). Then, the formula ϕ_σ is obtained as the conjunction of these subformulas computed as follows:

- Checking the first requirement: For each $i \in \{0, \ldots, K\}$ and for each $g_i, g_{i+1} \in G$, we compute a formula $\phi_{(i, g_i, g_{i+1})}$ such that $\phi_{(i, g_i, g_{i+1})}(U)$ is true iff there are $w_i, w_{i+1}' \in \Gamma \cup \{\varepsilon\}$ and $N_i \in M[(\Gamma \cup \{\varepsilon\}) \times \{i\}]$, and $N_{i+1}'' \in M[(\Gamma \cup \{\varepsilon\}) \times \{i+1\}]$ such that: (1) $\langle g_i, (w_i, i), N_i \rangle \Rightarrow_i^* \langle g_{i+1}, (w_{i+1}', i+1), N_{i+1}'' \rangle$, (2) $([(w_i, i)] + N_i)((w, i)) = U(x_{(w,i)})$, and (3) $([(w_{i+1}', i+1)] + N_{i+1}'')((w, i+1)) = U(y_{(w,i+1)})$ for all $w \in \Gamma \cup \{\varepsilon\}$.

To this end, we prove that the set of valuation $U : \mathcal{V} \to \mathbb{N}$, such that there is a computation $\langle g_i, (w_i, i), N_i \rangle \Rightarrow_i^* \langle g_{i+1}, (w_{i+1}', i+1), N_{i+1}'' \rangle$ of $\mathcal{A}$, where for every $w \in \Gamma \cup \{\varepsilon\}$, $([(w_i, i)] + N_i)((w, i)) = U(x_{(w,i)})$ and $([(w_{i+1}', i+1)] + N_{i+1}'')((w, i+1)) = U(y_{(w,i+1)})$, can be defined as the Parikh image of a language accepted by a finite state automaton $\mathcal{S}_{(i, g_i, g_{i+1})}$ with the input alphabet $(\Gamma \cup \{\varepsilon\}) \times \{i, i+1\}$. Then, we use Theorem 4 to construct the formula $\phi_{(i, g_i, g_{i+1})}$. The automaton $\mathcal{S}_{(i, g_i, g_{i+1})}$ has the following structure:

- The set of states:
  - $\mathcal{S}_{(i, g_i, g_{i+1})}$ has two special states: s^{init} as an initial state and s^{final} as a final state.
  - For each $g \in G$, the automaton $\mathcal{S}_{(i, g_i, g_{i+1})}$ has a state g^c. This represents that the current value of the global store of $\mathcal{A}$ is g when a context switch occurs.
  - For each $g \in G$ and for each $w \in \Gamma \cup \{\varepsilon\}$, the automaton $\mathcal{S}_{(i, g_i, g_{i+1})}$ has a state (g, w). This corresponds to the fact that the current value of the global store of $\mathcal{A}$ is g and the local configuration of the active thread is (w, i).

- The set of transitions:
  - **Initialization** For each $w_i \in \Gamma \cup \{\varepsilon\}$, the automaton $\mathcal{S}_{(i, g_i, g_{i+1})}$ has the transition $s^{init} \xrightarrow{(w_i, i)} (g_i, w_i)$. This transition corresponds to a guess of the local configuration of the first active thread (w_i, i).

- **Simulation of a transition:** For each rule $g\gamma \hookrightarrow g'w'$ (resp. $g\gamma \hookrightarrow g'w' \rhd \gamma'$) of $\mathcal{A}$, $\mathcal{S}_{(i,g_i,g_{i+1})}$ has the transition $(g,\gamma) \xrightarrow{(\gamma',i+1)} (g',w')$ (resp. $(g,\gamma) \xrightarrow{\varepsilon} (g',w')$).
- **Simulation of a context switch** For each state $g \in G$ and each pair of stack configurations w and w' in $\Gamma \cup \{\varepsilon\}$, the automaton $\mathcal{S}_{(i,g_i,g_{i+1})}$ has the transitions $(g,w) \xrightarrow{(w,i+1)} g^c$ and $g^c \xrightarrow{(w',i)} (g,w')$. These transitions simulate a context switch between two threads with local configurations (w,i) and (w',i).
- **End of the simulation:** For each $w \in \Gamma \cup \{\varepsilon\}$, the automaton $\mathcal{S}_{(i,g_i,g_{i+1})}$ has the transition $(g_{i+1},w) \xrightarrow{(w,i+1)} s^{final}$. This corresponds to the interruption of the thread with local configuration (w,i) and to the end of the simulation.

It can be checked that the size of the automaton $\mathcal{S}_{(i,g_i,g_{i+1})}$ is polynomial in the size of $\mathcal{A}$. The relation between $\mathcal{S}_{(i,g_i,g_{i+1})}$ and $\mathcal{A}$ is given by the following lemma:

Lemma 3. *A word u is in $L(\mathcal{S}_{(i,g_i,g_{i+1})})$ iff there are $w_i, w'_{i+1} \in (\Gamma \cup \{\varepsilon\})$, $N_i \in M[(\Gamma \cup \{\varepsilon\}) \times \{i\}]$, and $N''_{i+1} \in M[(\Gamma \cup \{\varepsilon\}) \times \{i+1\}]$ such that: (1) $\langle g_i,(w_i,i),N_i \rangle \Rightarrow^*_i \langle g_{i+1},(w'_{i+1},i+1),N''_{i+1} \rangle$, (2) $([(w_i,i)] + N_i)((w,i)) = [u]((w,i))$, and (3) $([(w'_{i+1},i+1)] + N''_{i+1})((w,i+1)) = [u]((w,i+1))$ for every stack configuration $w \in \Gamma \cup \{\varepsilon\}$.*

As a consequence of Theorem 4 and Lemma 3, it is possible to compute in polynomial time and size an existential Presburger formula $\phi_{(i,g_i,g_{i+1})}$ that represents the Parikh image of the language $L(\mathcal{S}_{(i,g_i,g_{i+1})})$.

Lemma 4. *For every valuation U, $\phi_{(i,g_i,g_{i+1})}(U)$ is true iff there are $w_i, w'_{i+1} \in (\Gamma \cup \{\varepsilon\})$, $N_i \in M[(\Gamma \cup \{\varepsilon\}) \times \{i\}]$, and $N''_{i+1} \in M[(\Gamma \cup \{\varepsilon\}) \times \{i+1\}]$ such that: (1) $\langle g_i,(w_i,i),N_i \rangle \Rightarrow^*_i \langle g_{i+1},(w'_{i+1},i+1),N''_{i+1} \rangle$, (2) $([(w_i,i)] + N_i)((w,i)) = U(x_{(w,i)})$, and (3) $([(w'_{i+1},i+1)] + N''_{i+1})((w,i+1)) = U(y_{(w,i+1)})$ for every $w \in \Gamma \cup \{\varepsilon\}$.*

- Checking the second requirement: The requirement that $N_i + [(w_i,i)] \leq N''_i + [(w'_i,i)]$, for every $i \in \{1,\ldots,K\}$, can be expressed by $\phi_{(i,2)} = \bigwedge_{w \in (\Gamma \cup \{\varepsilon\})} x_{(w,i)} \leq y_{(w,i)}$.

- Checking the third requirement: To express the requirements that $N_0 = N''_0 = \emptyset$ and $w_0 = w'_0 = \gamma_0$, we consider the formula ϕ_3 as the conjunction of the formulas: $x_{(\gamma_0,0)} = y_{(\gamma_0,0)} = 1$ and $x_{(w,0)} = y_{(w,0)} = 0$ for all $w \in \Gamma \cup \{\varepsilon\}$ such that $w \neq \gamma_0$.

Finally, the formula ϕ_σ is obtained as $\phi_3 \wedge \left(\bigwedge_{i \in \{0,\ldots,K\}} (\phi_{(i,g_i,g_{i+1})} \wedge \phi_{(i,2)}) \right)$. It can be checked that the size of ϕ_σ is polynomial in K and in the size of $\mathcal{A}$. $\qquad\square$

5   Analysis of Dynamic Networks of Concurrent Pushdown Systems

We consider now the case of DCPSs. It is well-known that the SRP is undecidable already for networks with two concurrent pushdown processes. We prove however that both problems SRP[K] and SRP[K,L] are decidable, for any given bounds K and L. For that, we prove the following fact.

Theorem 7. *For every $K,L \in \mathbb{N}$, the problems SRP[K] and the SRP[K,L] for DCPS are exponentially reducible to the corresponding problems for DCFS.*

The rest of this section is devoted to the proof of Theorem 7. Let us fix a DCPS $\mathcal{A} = (G,\Gamma,\Delta,g_0,\gamma_0)$. We show that it is possible to construct a DCFS $\mathcal{A}_{fs}$ such that

the problems SRP[K] and SRP[K, L] for $\mathcal{A}$ can be reduced to the corresponding problems for $\mathcal{A}_{fs}$. Let us present the main steps of this construction. For that, let us consider the problem SRP[K], for some fixed $K \in \mathbb{N}$. Then, let us concentrate on the computations of one thread, and assume that this thread will be interrupted and resumed i times (with $i \leq K$) during its execution from some initial global state g and initial local state γ to some final global state g'. The computations of such a thread correspond to the run of a labeled pushdown system, built out of $\mathcal{A}$, which (1) performs the same operations on the stack and global states as the ones specified by Δ, (2) makes visible as transition labels the local state (element of Γ) of the spawned threads, and (3) nondeterministically guesses jumps from a global state to another one corresponding to the effect of context switches. These jumps are also made visible as transition labels under the form of pairs $(g_l, g_{l+1}) \in G \times G$ (meaning that the computation of the thread is interrupted at state g_l and is resumed at state g_{l+1}). The number of such jumps in each run is precisely i.

Then, the problem is to handle the composition of all the computations (unbounded number) of the generated threads and to make sure that the guesses made by each one of them (on their control state jumps due to context switches) are correct. The key observation which allows to solve this problem is that it is possible to assume without loss of preciseness that some of the generated threads can be ignored (or lost). Indeed, these threads can always be considered as threads which will never be scheduled. Therefore, the behaviors of each thread can be modeled using a finite-state automaton which recognizes the downward closure of the language of the labeled pushdown system of a thread w.r.t. the ordering on words where u is less than v if u can be obtained from v by erasing symbol in Γ. We know by Theorem 1 that this automaton is effectively constructible. So, let $\mathcal{S}_{(i,g,\gamma,g')}$ be the automaton modeling the computations of threads starting from g and γ and reaching g' after i interruptions-resumptions.

The next step is to synchronize the so-defined finite-state automata in order to represent valid computations of the whole system. For that, we define a DCFS $\mathcal{A}_{fs}$ which simulates the composition of these automata as follows:

- Assume that the initial global state is g_0 and that the initial thread has an starting local state γ_0. Then, $\mathcal{A}_{fs}$ guesses for this thread the number of switches i and its final state g, and starts simulating its behaviors according to the transitions of the automaton $\mathcal{S}_{(i,g_0,\gamma_0,g)}$. To initialize the simulation, $\mathcal{A}_{fs}$ has a rule $g_0 \gamma_0 \hookrightarrow \$s^{init}_{(i,g_0,\gamma_0,g)}$, where $s^{init}_{(i,g_0,\gamma_0,g)}$ is the initial state of $\mathcal{S}_{(i,g_0,\gamma_0,g)}$. This rule allows to check that the control state is g_0 and to move to a special control state \$ corresponding to a simulation phase without context switches.

- During the simulation, when a transition $s \xrightarrow{\gamma} s'$ is encountered, a new thread γ is spawned by $\mathcal{A}_{fs}$. This is done using a rule $\$s \hookrightarrow \$s' \rhd \gamma$. The new thread will stay pending until $\mathcal{A}_{fs}$ can dispatch it.

- A pending thread γ which has never been activated can be dispatched by $\mathcal{A}_{fs}$ at the moment of a context switch. For that, $\mathcal{A}_{fs}$ has a rule $g\gamma \hookrightarrow \$s^{init}_{(i,g,\gamma,g')}$ where $s^{init}_{(i,g,\gamma,g')}$ is the initial state of $\mathcal{S}_{(i,g,\gamma,g')}$, for every possible starting and ending states g and g', and every possible number of context switches $i \leq K$.

- Encountering a transition $s \xrightarrow{(g_1,g_2)} s'$ means that the computation of the simulated thread has lead to the global store g_1, and that this computation will be interrupted

at this point and will be resumed later when the global store will become g_2 (due to the execution of some other threads). Then, $\mathcal{A}_{fs}$ moves from its control state $ to a control state g_1 so that the control can be taken by another thread (which was waiting for g_1), and transforms the local state of the current thread (which may be interrupted) to (g_2, s'). Both of these operations are done using a rule $\$s \hookrightarrow g_1(g_2, s')$. In the case of $g_1 \neq g_2$, we observe that the only action that can be done by $\mathcal{A}_{fs}$ after executing this rule is a context switch, i.e., (g_2, s') becomes idle and some pending thread is activated (either dispatched for the first time, or resumed after some interruption). We have seen above how $\mathcal{A}_{fs}$ dispatches pending threads for the first time. The resumption of threads at control state g_1 is done by having rules of the form $g_1(g_1, s'') \hookrightarrow \$s''$ for all possible states s'' in $\mathcal{S}_{(i,g,\gamma,g')}$. Such a rule means that if a pending thread (g_1, s'') exists, then it can be resumed and the simulation of its behaviors is pursued from the state s'' (at which it was stopped at the last interruption). Similarly, (g_2, s') will be resumed when the rule $g_2(g_2, s') \hookrightarrow \$s'$ can be executed which can only happen if the global store becomes g_2.

- Finally, when a final state $s_{(i,g,\gamma,g')}^{final}$ of $\mathcal{S}_{(i,g,\gamma,g')}$ is reached, this means that the simulation of the current thread has been completed and therefore the global store must be g' (the guessed target state) at this point. Then, the execution of the rule $\$s_{(i,g,\gamma,g')}^{final} \hookrightarrow g'\bot$ allows to release the control so that some pending thread waiting for g' can be resumed.

Let us give in more details the construction described above.

5.1   Simulating Threads with Finite-State Automata

First, we define the DCPS $\mathcal{A}_{los}$ obtained from $\mathcal{A}$ by allowing losses of the generated threads. Let $\mathcal{A}_{los} = (G, \Gamma, \Delta_{los}, g_0, \gamma_0)$ be the DCPS such that $\Delta_{los} = \Delta \cup \{g\gamma \hookrightarrow g'w' \mid (g\gamma \hookrightarrow g'w' \rhd \gamma') \in \Delta\}$.

Lemma 5. *For every $K, L \in \mathbb{N}$, a control state $g \in G$ is K-reachable (resp. $[K,L]$-reachable) in $\mathcal{A}$ if and only if g is K-reachable (resp. $[K,L]$-reachable) in $\mathcal{A}_{los}$.*

Next, we show the construction of the automaton $\mathcal{S}_{(i,g,\gamma,g')}$ for some given $i \in \{0, \ldots, K\}$, $\gamma \in \Gamma$, and $g, g' \in G$. For that, we start by considering a labeled pushdown system simulating the behaviors of thread that reaches the state g' starting from g and the stack configuration γ after some number of jumps in the control state (representing guesses on the effect of context switches), The spawned thread as well as the guesses on the control state jumps made during the computation are made visible as labels on the transitions. Let $\mathcal{P} = (G, \Gamma \cup G \times G, \Gamma, \Delta_{\mathcal{P}})$ be the labeled pushdown system where $\Delta_{\mathcal{P}}$ is the smallest set of rule such that (1) for every $g_1\gamma_1 \hookrightarrow g_2w \rhd \gamma_2$ (resp. $g_1\gamma_1 \hookrightarrow g_2w$) in Δ_{los}, the rule $g_1\gamma_1 \overset{\gamma_2}{\hookrightarrow} g_2w$ (resp. $g_1\gamma_1 \overset{\varepsilon}{\hookrightarrow} g_2w$) is in $\Delta_{\mathcal{P}}$, and (2) for every $(g_1, g_2) \in G \times G$, and for every $\gamma_1 \in \Gamma$, the rule $g_1\gamma_1 \overset{(g_1,g_2)}{\hookrightarrow} g_2\gamma_1$ is in $\Delta_{\mathcal{P}}$.

Then, the set of behaviors represented by this labeled pushdown system which correspond to precisely i control switches (interruption-resumptions) is

$$L_{(i,g,\gamma,g')} = L_{\mathcal{P}}(g\gamma, g') \cap (\Gamma^* \cdot (G \times G) \cdot \Gamma^*)^i.$$

This set is context-free in general (since it is the intersection of a context-free language with a regular one). However, due to Lemma 5, we can consider without loss of preciseness the downward closure of $L_{(i,g,\gamma,g')}$ w.r.t. the sub-word relation corresponding to the deletion of symbols in Γ while preserving all symbols in $G \times G$, i.e., the set

$$L_{(i,g,\gamma,g')} \downarrow \cap (\Gamma^* \cdot (G \times G) \cdot \Gamma^*)^i.$$

By Theorem 1, this set regular an can be effectively represented by a finite-state automaton $\mathcal{S}_{(i,g,\gamma,g')} = (S_{(i,g,\gamma,g')}, \Gamma \cup G \times G, \delta_{(i,g,\gamma,g')}, s^{init}_{(i,g,\gamma,g')}, s^{final}_{(i,g,\gamma,g')})$. We assume w.l.o.g that all states in the automaton $\mathcal{S}_{(i,g,\gamma,g')}$ are co-reachable from the final state.

Lemma 6. *Let $i \in \mathbb{N}$ be a natural number, $\gamma \in \Gamma$ be a stack symbol, $g_1, g_1', g_2, \ldots, g_{i+1}, g_{i+1}' \in G$ be a sequence of states, and $w_0, \ldots, w_i \in \Gamma^*$ be a sequence of stack contents. Then, $w_0(g_1', g_2) w_1 (g_2', g_3) w_2 \cdots (g_i', g_{i+1}) w_i$ is accepted by $\mathcal{S}_{(i,g_1,\gamma,g_{i+1}')}$ iff there are $u_0, \ldots, u_{i+1} \in \Gamma^*$ such that, for every $l \in \{1, \ldots, i+1\}$ and $j \in \mathbb{N}$, $\langle g_l, (u_{l-1}, j), \emptyset \rangle \Rightarrow^*_j \langle g_l', (u_l, j), M_{l-1} \rangle$ is a computation of $\mathcal{A}_{los}$ where $u_0 = \gamma$ and $M_{l-1}((\gamma', j+1)) = [w_{l-1}](\gamma')$ for all $\gamma' \in \Gamma$.*

The lemma above says that a word $w_0(g_1', g_2) w_1 (g_2', g_3) w_2 \cdots (g_i', g_{i+1}) w_i$ is in $L(\mathcal{S}_{(i,g_1,\gamma,g_{i+1}')})$ iff the DCPS $\mathcal{A}_{los}$ is able to bring the value of the global variables from g_l to g_l' and the stack configuration of the simulated thread from u_{l-1} to u_l while creating the set of threads with initial stack symbols in $[w_{l-1}]$ for every $l \in \{1, \ldots, i+1\}$.

5.2   From DCPS to DCFS

We define the DCFS $\mathcal{A}_{fs} = (G_{fs}, \Gamma_{fs}, \Delta_{fs}, g_0, \gamma_0)$ where:

- $G_{fs} = G \cup \{\$\}$ is a finite set of states with $\$ \notin G$.
- Γ_{fs} is a finite set of stack alphabet defined as the union of the sets $\Gamma \cup \{\bot\}$, $S_{(i,g,\gamma,g')}$ and $G \times S_{(i,g,\gamma,g')}$ for all $(i,g,\gamma,g') \in \{0, \ldots, K\} \times G \times \Gamma \times G$, where $S_{(i,g,\gamma,g')}$ is the set of states of $\mathcal{S}_{(i,g,\gamma,g')}$.
- Δ_{fs} is the smallest set of transitions such that
  - *Initialize/Disptach:* For every $i \in \{0, \ldots, K\}$, every $\gamma \in \Gamma$, and every $g, g' \in G$, the rule $g\gamma \hookrightarrow \$s^{init}_{(i,g,\gamma,g')}$ is in Δ_{fs} where $s^{init}_{(i,g,\gamma,g')}$ is the initial state of $\mathcal{S}_{(i,g,\gamma,g')}$.
  - *Skip:* For every transition $s \xrightarrow{\varepsilon} s'$ of $\mathcal{S}_{(i,g,\gamma,g')}$, the rule $\$s \hookrightarrow \$s'$ is in Δ_{fs}.
  - *Spawn:* For every transition $s \xrightarrow{\gamma} s'$ of $\mathcal{S}_{(i,g,\gamma,g')}$, the rule $\$s \hookrightarrow \$s' \rhd \gamma$ is in Δ_{fs}.
  - *Interrupt:* For every transition $s \xrightarrow{(g_1,g_2)} s'$ of $\mathcal{S}_{(i,g,\gamma,g')}$, the rule $\$s \hookrightarrow g_1(g_2, s')$ is in Δ_{fs}.
  - *Resume:* For every $(g, s) \in \Gamma_{fs}$, the rule $g(g, s) \hookrightarrow \$s$ is in Δ_{fs}.
  - *Terminate:* The rule $\$s^{final}_{(i,g,\gamma,g')} \hookrightarrow g'\bot$ is in Δ_{fs}, where $s^{final}_{(i,g,\gamma,g')}$ is the final state of $\mathcal{S}_{(i,g,\gamma,g')}$.

Lemma 7. *For every $K, L \in \mathbb{N}$, a control state g is K-reachable (resp. $[K,L]$-reachable) in $\mathcal{A}$ if and only if g is K-reachable (resp. $[K,L]$-reachable) in $\mathcal{A}_{fs}$.*

The proof of the lemma above is technical and is given in details in [1]. We give hereafter a high level description of it. Let us consider a DCPS $\mathcal{A}_+$ which is *the union* of

$\mathcal{A}_{los}$ and $\mathcal{A}_{fs}$ in the sense that for each created thread with initial configuration $\gamma \in \Gamma$, $\mathcal{A}_+$ chooses nondeterministically whether the thread will be executed according to the rules of $\mathcal{A}_{los}$, or simulated according to the rules of $\mathcal{A}_{fs}$. Then, we define the rank of a computation of $\mathcal{A}_+$ to be the pair $(m,n) \in \mathbb{N} \times \mathbb{N}$ where m is the number of threads involved in the computation that follow the rules of $\mathcal{A}_{los}$ and n is the number of threads in the computation following the rules of $\mathcal{A}_{fs}$. Observe that computations of rank (m,n) where $n=0$ (resp. $m=0$) are precisely the computations of $\mathcal{A}_{los}$ (resp. $\mathcal{A}_{fs}$). We prove that for any computation of $\mathcal{A}_+$ of rank $(m+1,n)$ (resp. $(m,n+1)$), there exists a computations of $\mathcal{A}_+$ of rank $(m,n+1)$ (resp. $(m+1,n)$). This computation is obtained from the original one by simulating the execution of a thread that follows the rules of $\mathcal{A}_{los}$ (resp. $\mathcal{A}_{fs}$) by a thread that follows the rules of $\mathcal{A}_{fs}$ (resp. $\mathcal{A}_{los}$). This is possible thanks to Lemma 6. A consequence of this fact is that, for every $m \in \mathbb{N}$, a control state is K-reachable (resp. $[K,L]$-reachable) by a computation of rank $(m,0)$ (i.e., by a computation of $\mathcal{A}_{los}$ with m threads) if and only if it is K-reachable (resp. $[K,L]$-reachable) by a computation of rank $(0,m)$ (i.e. by a computation of $\mathcal{A}_{fs}$ with the m threads). This is precisely what Lemma 7 is saying.

Finally, Theorem 7 is an immediate consequence of Lemma 7. A corollary of Theorem 7 and Theorem 5 is the following fact.

Corollary 1. *For every $K \in \mathbb{N}$, the problem SRP$[K]$ for DCPS is in 2-EXPSPACE, and for every $K,L \in \mathbb{N}$, the problem SRP$[K,L]$ for DCPS is in NEXPTIME.*

6   Conclusion

We have proposed new concepts for context-bounded verification we believe that are natural and suitable for programs with dynamic thread creation. These concepts are based on the idea of bounding the number of switches for each thread and not for all the threads in a computation.

First, we have proved that even for finite-state threads, adopting such a notion of context-bounding leads in general to a problem which is as hard as the coverability problem of Petri nets. This means that, in theory, the complexity of this problem is high, but in practice, there are quite efficient techniques (based on iterative computation of under/upper approximations) developed recently for solving this problem which have been implemented and used successfully in [5,4]. Moreover, we have proposed a notion of stratified context-bounding for which the verification is in NP, i.e., as hard as in the case without dynamic thread creation. An interesting question is how to implement efficiently the analysis in this case using clever encodings in SMT solvers.

Moreover, we have proved that the considered problems are still decidable for the case of pushdown threads. This is done by a nontrivial reduction to the corresponding problems for finite-state threads. This reduction is based on computing the regular downward closure of context-free languages w.r.t. the sub-word relation. The downward closure computation may lead in general to an unavoidable exponential blow-up. This is due to the succinctness of context-free grammars w.r.t. finite state automata: For instance, the finite language $\{a^{2^N}\}$, for a fixed $N \geq 1$, can be defined with a context-free grammar of size N whereas a finite-state automaton representing it (or its downward

closure) is necessarily of size at least 2^N. An interesting open problem is whether there is an alternative proof technique which allows to avoid the downward closure construction. In practice, we believe that it would be possible to overcome this problem by for instance designing algorithms allowing to generate efficiently and incrementally (parts of the) downward closure.

Finally, in our models, we consider that each created thread inherits a switch number from its father (the one of its father plus 1). An alternative definition can be obtained by considering that each created thread is given the switch number 0. (Therefore, each thread can perform up to K switches.) For that model, we can prove (for more details see [1]) that our results concerning the reachability problems SRP and SRP[K] hold with the same complexity bounds. However, the problem SRP[K, L] for finite state threads (resp. pushdown threads) becomes EXPSPACE-complete (in 2-EXPSPACE) instead of NP-complete (NEXPTIME) for this definition.

References

1. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. Technical report, LIAFA (October 2008)
2. Bouajjani, A., Esparza, J., Schwoon, S., Strejcek, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
3. Courcelle, B.: On construction obstruction sets of words. In: EATCS 1991, vol. 44, pp. 178–185 (1991)
4. Ganty, P., Raskin, J.F., Begin, L.V.: A complete abstract interpretation framework for coverability properties of WSTS. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg (2005)
5. Geeraerts, G., Raskin, J.F., Begin, L.V.: Expand, enlarge and check: New algorithms for the coverability problem of WSTS. J. Comput. Syst. Sci. 72(1), 180–203 (2006)
6. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
7. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
8. Lipton, R.: The reachability problem requires exponential time. Technical Report TR 66 (1976)
9. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI 2004, pp. 446–455. ACM Press, New York (2007)
10. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
11. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI 2004, pp. 14–24. ACM, New York (2004)
12. Rackoff, C.: The covering and boundedness problem for vector addition systems. Theoretical Computer Science (1978)
13. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)

14. Seidl, H., Schwentick, T., Muscholl, A., Habermehl, P.: Counting in trees for free. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1136–1149. Springer, Heidelberg (2004)
15. Stewart, I.A.: Reachability in some classes of acyclic petri nets. Fundam. Inform. 23(1), 91–100 (1995)
16. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic context-bounded analysis of multithreaded java programs. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)
17. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational Horn clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 337–352. Springer, Heidelberg (2005)
18. Zaks, A., Joshi, R.: Verifying multi-threaded C programs with SPIN. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 325–342. Springer, Heidelberg (2008)

Semantic Reduction of Thread Interleavings in Concurrent Programs

Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta

NEC Laboratories America, 4 Independence Way, Princeton, NJ
{kahlon,srirams,agupta}@nec-labs.com

Abstract. We propose a static analysis framework for concurrent programs based on reduction of thread interleavings using sound invariants on the top of partial order techniques. Starting from a product graph that represents transactions, we iteratively refine the graph to remove statically unreachable nodes in the product graph using the results of these analyses. We use abstract interpretation to automatically derive program invariants, based on abstract domains of increasing precision. We demonstrate the benefits of this framework in an application to find data race bugs in concurrent programs, where our static analyses serve to reduce the number of false warnings captured by an initial lockset analysis. This framework also facilitates use of model checking on the remaining warnings to generate concrete error traces, where we leverage the preceding static analyses to generate small program slices and the derived invariants to improve scalability. We describe our experimental results on a suite of Linux device drivers.

1 Introduction

Concrete error traces are critical for effective debugging of software. Unfortunately, generating error traces for concurrency related bugs is notoriously hard. One of the key reasons for this is that concurrent programs are behaviorally complex due to the many possible interleavings between threads. These interleavings make concurrent programs hard to analyze.

Verification and analysis for concurrent systems is currently a very active area of research due to the multi-core revolution. Testing, static analysis, and model checking have all been explored but not without some drawbacks. Testing has clearly been the most effective debugging technique for sequential programs. However, the multitude of interleavings among threads makes it hard to provide meaningful coverage guarantees for concurrent programs. Furthermore, replayability of the bugs detected through testing is a challenge.

Static analysis techniques have been successful for detecting standard concurrency bugs such as data races and deadlocks [28,25,24,11,26,15]. However, the large number of bogus warnings generated by static analyzers remains a drawback. Most static analyses ignore conditional statements, focusing mostly on the syntactic reachability of a pair of control locations rather than semantic reachability. This places the burden of sifting the true bugs from the false warnings on the programmer, which leads to poor productivity.

Model checking [5,3,6] has the advantages of systematic state space exploration, and can produce concrete error traces. However, the state explosion problem severely limits its scalability on large real-life concurrent programs.

In this paper, we propose a general framework for analysis of concurrent programs, based on semantic reduction in the thread interleavings by use of sound invariants. We first utilize *partial order reduction* (POR) techniques and constraints from synchronization primitives to construct a *transaction graph*. The transaction graph effectively captures the relevant thread interleavings for performing a sound static analysis. We then derive *sound invariants* by using *abstract interpretation* over this transaction graph. These invariants are used to further refine the transaction graph by removing unreachable nodes. This can lead to larger transactions, i.e. a reduction in the thread interleavings. The removal also facilitates the discovery of stronger invariants on the reduced graph. This process can be iterated until convergence, i.e. until no more nodes can be removed and no better invariants computed. The transaction graph is central in our approach: at any stage it provides a current snapshot of thread interleavings needed for sound static analysis. It also allows us to stage various analyses to achieve scalability, such that less precise but cheaper techniques are used first to refine the transaction graph, to enable application of more precise techniques later. Our focus is on successive reduction of thread interleavings, which is a primary source of complexity in the analysis of concurrent programs.

```
a0:  void Alloc_Page() {
a1:   a  := c;
a2:   pt_lock(&plk);
a3:   if  (pg_cnt ≥ LIMIT)  {
a4:        pt_wait(&pg_lim,&plk);
a5:        incr (pg_count) ;
a6:        pt_unlock(&plk);
a7:        sh1  := sh ;
a8:   } else {
a9:        pt_lock(&count_lock);
a10:       pt_unlock(&plk);
a11:       page  := alloc_page();
a12:       sh  := 5 ;
a13:       if (page)
a14:            incr( pg_count );
a15:       pt_unlock(&count_lock);
a16: end-if
a17: b  := a +1;
a18: end-function
```

```
b0:  void Dealloc_Page() {
b1:   pt_lock(&plk);
b2:   if  (pg_count == LIMIT)
b3:        sh  := 2 ;
b4:        decr (pg_count) ;
b5:        sh1  := sh ;
b6:        pt_notify(& pg_lim, & plk);
b7:        pt_unlock(& plk);
b8:   } else {
b9:        pt_lock(&count_lock);
b10:       pt_unlock(&plk);
b11:       decr(pg_count);
b12:       sh  := 4 ;
b13:       pt_unlock(&count_lock);
b14: end-if
b15: end-function
```

Fig. 1. Example concurrent program consisting of two functions

Motivating Example

The concurrent program shown in Fig. 1 comprises of *multiple threads* executing the *Alloc_Page* and *Dealloc_Page* routines. We assume that each statement shown is executed atomically. For clarity, all shared variable accesses and synchronization constructs **pt_lock**, **pt_unlock**, **pt_wait** and **pt_notify** are highlighted.

The shared variable *sh* is written to at location **a**12, **b**3 and **b**12. Since the set of locks held at **a12** and **b3** (viz., $\{count\_lk\}$ and $\{plk\}$, respectively) are disjoint, the pair (**a**12, **b**3) normally constitutes a data race warning according to lockset-based race detection techniques.

When *some thread* reaches the control location **b**3, we automatically establish the interval invariant ψ_3 : **pg_count** $\in [\text{LIMIT}, +\infty)$, regardless of the control location of the other threads. This invariant is quite challenging to establish (the reader is invited to attempt). Specifically, establishing ψ_3 requires us to reason about the sequence of synchronizations between threads as well as the conditional branches involved.

Likewise, we establish the invariant φ_{12} : **pg_count** $\in (-\infty, \text{LIMIT})$ when *some thread* reaches the control location **a**12, regardless of the location of the other threads. Once again, this invariant is non-trivial to establish.It involves conditional branches as well as thread synchronization.

Using the invariants computed, we conclude that locations **a**12 and **b**3 are *not simultaneously reachable*. In other words, the lockset-based method yields a bogus warning that can be eliminated by means of the automatically computed invariants ψ_3, φ_{12}.

2   Program Model

We consider concurrent imperative programs comprising threads that communicate using shared variables and synchronize with each other using standard primitives such as locks, rendezvous, etc.

Program Representation. Each thread in a concurrent program $T_i : \langle F_i, \mathsf{e}_i, G_i, L_i \rangle$ consists of procedures F_i, entry procedure $\mathsf{e}_i \in F_i$, a set of global variables G and thread local variables v. Each procedure $p \in F$, is associated with a tuple of formal arguments $\mathsf{args}(p)$, a return type t_p, local variables $L(p)$, and a control flow graph (CFG). Each procedural CFG $\langle N(p), E(p), \text{action} \rangle$ consists of a set of nodes $N(p)$ and a set of edges $E(p)$ between nodes in $N(p)$. Each edge $m \to n \in E(p)$ is associated with an *action* that is an assignment, a call to another procedure, a return statement, a conditional guard, or a synchronization statement. The actions in the CFG for a procedure p may refer to variables in the set $G \cup \mathsf{args}(p) \cup L(p)$.

A multi-threaded program Π consists of a set of threads $T_1, \ldots, T_N$ for some fixed $N > 0$ and a set of shared variables S. Note that every shared variable $s \in S$ is a global variable in each thread T_i.

Threads synchronize with each other using standard primitives like locks, rendezvous and broadcasts. Locks are standard primitives used to enforce mutually exclusive access to shared resources. They occur very commonly in many parallel

programming paradigms and are widely used to enforce thread synchronization. Rendezvous are motivated by `wait/notify` primitives of Java and condition variables in the `POSIX` thread library. Rendezvous find limited use in applications such as browsers, device drivers and scientific programs. Broadcasts are seldom seen in practice.

2.1   Preliminaries

Static program analysis can be used to compute sound invariant assertions that characterize the values of program variables at different program points. Since our approach involves reasoning with infinite sets of integers and reals, the *abstract interpretation* framework forms an integral part. We provide a concise description of abstract interpretation in this section. A detailed presentation is available elsewhere [9,8].

Let Ψ be a CFG of a sequential (single threaded) program. Concurrent programs are treated in Section 4. For simplicity, we assume that Ψ consists of a single procedure that does not involve calls to other procedures. Procedures, including recursive procedures, can be handled by implementing a context-sensitive program analysis. All variables involved in Ψ are assumed to be integers. Pointers and arrays are lowered into integers using a process of memory modeling followed by abstraction. Such an abstraction is implemented by the F-Soft framework [14]. As mentioned earlier, each edge in the CFG is labeled with an assignment or a condition.

An *abstract domain* Γ consists of assertions drawn from a selected assertion language which form a lattice through the partial order $\sqsubseteq$ modeling logical inclusion between assertions. Each object $a \in \Gamma$ represents a set of program states $[[a]]$. For the analysis, we require the following operations to be defined over Γ:

(a) Join: Given $a_1, a_2 \in \Gamma$, the join $a = a_1 \sqcup a_2$ is the smallest abstract object a w.r.t $\sqcup$ such that $a_1 \sqsubseteq a$, $a_2 \sqsubseteq a$.
(b) *Meet:* The meet $a_1 \sqcap a_2$ corresponds to the logical conjunction.
(c) *Abstract post condition post$_\Gamma$* models the effect of assignments.
(d) *Inclusion test* $\sqsubseteq$ to check for the termination.
(e) *Widening* operator ∇ to force convergence for the program loops.
(f) *Projection* operator $\exists$ removes out-of-scope variables.
(g) *Narrowing* operator $\triangle$ is used for solution improvement.

Given a program Ψ and an abstract domain Γ, we seek a map $\pi : L \mapsto \Gamma$ that maps each CFG location $\ell \in L$ to an abstract object $\pi(\ell)$. Such a map is constructed iteratively by the *forward propagation* iteration used in data-flow analysis:

$$\pi^0(\ell) = \begin{cases} \top, & \text{if } \ell = \ell_0 \\ \bot, & \text{otherwise} \end{cases} \text{ and } \pi^{i+1}(\ell) = \bigsqcup_{e: \ m \to \ell} post_\Gamma(\pi^i(m), e).$$

If the iteration converges, i.e, $\pi^{i+1}(\ell) \sqsubseteq \pi^i(\ell)$ for all $\ell \in L$ for some $i > 0$, π^{i+1} is the result of our analysis. However, unless the lattice Γ is of finite height or satisfies the *ascending chain condition*, convergence is not always guaranteed. On the

other hand, many of the domains commonly used in verification do not exhibit these conditions. Convergence is forced by using *widening* and *narrowing* [9].

Using abstract interpretation, we may lift dataflow analyses to semantically rich domains such as *intervals*, *polyhedra*, *shape graphs* and other domains to verify sophisticated, data-intensive properties. Intervals, Octagons and Polyhedra are instances of numerical domains that may be used to reason about the numerical operations in the program.

The interval domain consists of assertions of the form $x_i \in [\ell, u]$, associating each variable with an interval containing its possible values. The domain operations for the interval domain such as join, meet, post condition, inclusion, etc. can be performed efficiently (see [7] for details). However, the interval domain is *non-relational*. It computes an interval for each variable that is independent of the intervals for the other variables. As a result, it may fail to handle many commonly occurring situations that require more complex, relational invariants. The polyhedral domain [10] computes expressive linear invariants and is quite powerful. However, this power comes at the cost of having exponential time domain operations such as *post condition*, *join*, *projection* and so on.

The octagon domain due to Miné [22] extends the interval domain by computing intervals over program expressions such as $x - y$, $x + y$ and so on, for all possible pairs of program variables. The domain can perform operations such as post, join and projection efficiently using a graphical representation of the constraints and a canonical form based on shortest-path algorithms.

3   Transaction Graphs

In this section, we describe how we capture thread interleavings in the form of a transaction graph. We also describe our procedure for constructing an initial transaction graph by utilizing partial order reduction techniques and constraints due to synchronization primitives. The transaction graph will be used as a basis for the static analysis technique to be presented in the next section.

Let $\mathcal{P}$ be a concurrent program comprised of threads $T_1, ..., T_n$ and let N_i and E_i be the set of control locations and transitions of the control flow graph (CFG) of T_i, respectively. We write $\ell_i \rightsquigarrow m_i$ to denote a path from l_i to m_i.

Definition 1 (Transaction Graph). *A transaction graph $\Pi_\mathcal{P}$ of $\mathcal{P}$ is defined as $\Pi_\mathcal{P} = (N_\mathcal{P}, E_\mathcal{P})$, where $N_\mathcal{P} \subseteq N_1 \times ... \times N_n$ and $E_\mathcal{P} \subseteq N_\mathcal{P} \times N_\mathcal{P}$. Each edge of Π_P represents the execution of a sequence of statements by a thread T_i. Specifically, an edge is of the form $(l_1, \dots, l_i, \dots, l_n) \rightarrow (l_1, \dots, m_i, \dots, l_n)$, such that there is a path in T_i from $l_i \rightsquigarrow m_i$. Such an edge represents an execution of a program segment in thread T_i.*

A transaction graph considers a subset of the possible tuples of control states concurrently reachable by n different threads. Edges of the graph consist of a sequence of moves by a single thread. The *product graph* is a transaction graph $\Pi_\mathcal{P} \equiv \otimes_i N_i$ consisting of the *cartesian product* of the control locations in each thread, and each edge representing the execution of a single statement by a single

thread. In the presence of rendezvous and broadcast actions on the edges, the definition may be updated to necessitate synchronous rendezvous to happen in two consecutive moves.

The cartesian product graph consists of a superset of all the concurrent control states that need to be considered for the analysis of a given program. It is reduced to a transaction graph of manageable size by using partial order reduction (POR) [13] wherein we remove *redundant* control states that produce a result consistent with some other interleaving (see below).

Shared Variable Identification: The first step towards building a transaction graph consists of automatically and conservatively identifying shared variables. In general, shared variables are either global variables of threads, aliases thereof and pointers passed as parameters to API functions. Since global variables can be accessed via local pointers, alias analysis is key for identifying shared variables.

Our shared variable detection technique uses *update sequences*(Cf. [16]) to track aliasing information as well as whether a variable is being shared. An update sequence of assignments from pointers p to q along a sequence $\{l_j\}$ of consecutive thread locations is of the form $l_0 : p_1 = p, l_1 : p_2 = q_1, ..., l_{i-1} : p_i = q_{i-1}, l_i : p_{i+1} = q_i, ..., l_k : p = q_k$, where q_i is aliased to p_i between locations l_{i-1} and l_i. Then p is aliased to a shared variable if there exists an update sequence starting at a global variable or an escaped variable. Update sequences can be tracked via a simple and efficient dataflow analysis (see [17] and [16] for details).

Static Partial Order Techniques: The transaction graph is computed using partial order reduction (POR) which has been used extensively in model checking [13]. POR exploits the fact that concurrent computations are partial orders on operations of threads on shared variables. Therefore, instead of exploring all interleavings that realize this partial order, it suffices to explore just a few (ideally just one).

In this setting, we use POR over the complete product graph $\mathcal{P}$ of the thread CFGs, instead of over the state space of the concurrent program. We consider a pair of statements st_1 and st_2 of threads T_1 and T_2, respectively, to be *dependent* if (i) st_1 and st_2 access a common shared variable (including variables used for synchronization), and (ii) at least one of the accesses is a write operation. Whereas, this is a purely syntactic characterization, we may use semantic considerations to obtain a more refined dependence relation.

When constructing the transaction graph, a key goal is to maximize the lengths of the resulting transactions in the presence of scheduling constraints imposed by synchronization primitives. The general problem of delineating transactions of maximal length for threads synchronizing via locks and rendezvous is presented elsewhere [18]. For completeness, we present a simple transaction delineation algorithm for the easier case of two threads synchronizing via locks. Our algorithm is broadly similar to that of Godefroid [13] and is shown in Alg. 1. It is a worklist algorithm that traverses through the global control states of the concurrent program and computes their successors. The pair (l_1, l_2) represents

---

Algorithm 1. Transaction Delineation based on Static POR

1: Initialize $W = \{(in_1, in_2)\}$, where in_j is the initial state of thread T_j, and $Processed$ to $\emptyset$.
2: **repeat**
3:     Remove a state (l_1, l_2) from W and add it to $Processed$
4:     **if** neither l_1 nor l_2 is a shared object access **then**
5:         let $Succ_i = \{(m_1, m_2)|$ where (a) $m_{i'} = l_{i'}$ with $i' \in \{1, 2\}$ and $i' \neq i$, and (b) m_i is CFL-reachable from l_i via a local path x of thread T_i such that m_i is the first shared object access encountered along $x\}$
6:         Set $Succ = Succ_1 \cup Succ_2$
7:     **else if** l_1 is a shared object access of sh, say, **then**
8:         **if** m_2 is a statement accessing sh that is CFL-reachable from l_2 via a path x of T_2 such that (a) l_1 conflicts with m_2, and (b) no lock held at l_1 is acquired (and possibly released) along x **then**
9:             Let $Succ_1 = \{(m_1, l_2)|$ where m_1 is CFL-reachable from l_1 via a path y such that m_1 is the first shared object access along y after $l_1\}$
10:            Let $Succ_2 = \{(l_1, m_2)\}$
11:            Set $Succ = Succ_1 \cup Succ_2$
12:        **else**
13:            $Succ = \{(m_1, l_2)|$ m_1 is CFL-reachable from l_1 via a path x such that m_1 is the first shared object access along x after $l_1\}$
14:        **end if**
15:    **else if** l_2 is a shared object access of sh, say, **then**
16:        compute $Succ$ as in steps 7-14 with the roles of l_1 and l_2 reversed
17:    **end if**
18:    Add all states of $Succ$ not in $Processed$ to W.
19: **until** W is empty

---

a global control state in which threads T_1 and T_2 are at control locations l_1 and l_2, respectively.

At the initial state (in_1, in_2) of the given concurrent program, we let each thread execute until it encounters its first shared object access (steps 4-6). Next, in order to compute the successors of a global state (l_1, l_2) we need to decide whether a context switch is required at location l_i of thread T_i. A conflict analysis is carried out to determine whether T_1 is currently accessing a shared object sh at the location l_1 and whether thread T_2 starting at l_2 can reach a location m_2 which accesses sh and is dependent with l_1, i.e., l_1 conflicts with m_2. If so, then we explore interleavings wherein T_1 executes l_1 first leading to the set of successors $Succ_1$ (step 9) and those wherein T_2 executes the path leading to m_2 before l_1 is executed leading to the set of successors $Succ_2$ (step 10). On the other hand, if the synchronization primitives ensure that no path starting at l_2 leads to T_2 accessing sh then no context switch is required at l_1 and the successors of (l_1, l_2) result only from executing transitions of T_1 (step 13).

A crucial difference between Alg. 1 and the classical POR algorithm is that in deducing reachability of m_i from l_i we need to take recursion into account, i.e., we need to deduce CFL-reachability of m_i from l_i in thread T_i.
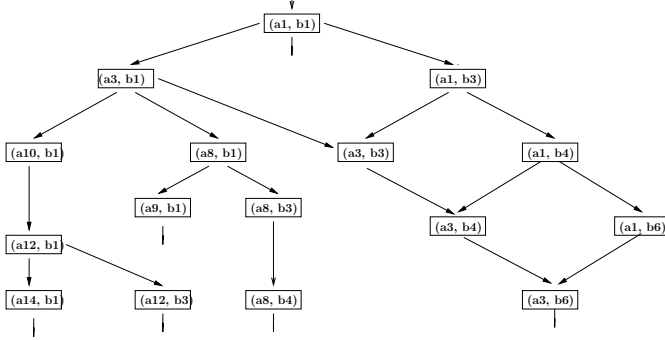
Fig. 2. Transaction Graph

Example 1. In Figure 1, the statements at locations **a**12 and **b**12 are dependent since the outcome for *sh* differs based on their relative order of execution. Statements **b**5 and **a**14 are independent, the actions performed by them commute. Statements **b**4 and **b**11 of Fig. 1 both decrease a shared variable pg_count by 1. Semantically, these statements are independent, since their order of execution does not affect the reachable states.

Synchronization Constraints. We now illustrate utilization of synchronization constraints in our running example from Figure 1. Fig. 2 shows a portion of the transaction graph obtained after pruning nodes with non-disjoint locksets. Examples of tuples pruned for non-disjoint lockset include (**a**5,**b**4), (**a**6,**b**3), (**a**3,**b**2) and so on which are mutually excluded by the lock plk. Note that a static lockset analysis on each thread allows us to avoid generating such pairs in the first place.

The send and wait statements **b**6 and **a**4, respectively, enforce that in a 2−threaded execution, **b**7 must be executed before **a**4. Therefore all the nodes in the set {(**a**5, **b**7), (**a**6, **b**7), (**a**7, **b**7)} are all unreachable. Again, computing the synchronization language at each location visited by each thread will ensure that such tuples are never considered.

The POR and synchronization-based constraints yield transactional sequence of actions for each thread so that an interleaving with another thread *need not* be considered during the execution of this sequence, while still capturing all the feasible interleavings. However, such sequences may be conditional on the location of the other thread. For instance, statements {**b**1, . . . ,**b**7 } are transactional provided the Alloc_Page thread resides in one of the locations {**a**1,. . . ,**a**7}. This allows us to *compact* many locations in the product graph into one single location in the initial transaction graph.

To summarize, partial order techniques and synchronization constraints are used to construct a transaction graph over the global control states. Although our presentation described these steps separately, our implementation derives an initial set of thread conflicts based on both considerations, which drives the construction of the transaction graph. The initial transaction graph is used to compute sound invariants, described in the next section.

4   Generation of Sound Invariants

We now apply abstract interpretation over the initial transaction graph to compute sound invariants. The key difference between abstract interpretation for sequential programs as opposed to transaction graphs, is the treatment of invariants that relate thread-local variables to shared variables. Such invariants are quite useful, since threads typically perform many actions involving locals and globals in an atomic section. However, local variables of a thread T_1 may not be shared directly with another thread T_2. A general solution is to consider a cartesian product of the abstract domain with itself, yielding tuples of invariant facts $\langle \varphi_1, \ldots, \varphi_n \rangle$ for each node of the transaction graph, wherein $\varphi_i[S, L_i]$ relates the global variables S with the local variables L_i of the ith thread.

For the sake of simplicity, we restrict our attention to two threads (i.e, $n = 2$). The invariant tuple annotating a node $\langle \ell_i, m_j \rangle$ of the transaction graph is denoted $\langle \varphi_i, \psi_j \rangle$, wherein $\varphi_i[S, L_1]$ relates the values of the shared and local variables for thread T_1 and $\psi_j[S, L_2]$ for thread T_2. Since the programs communicate through shared variables, we require that the set of shared variables described by any pair φ_i, ψ_j are the same: $(\exists L_1) \ \varphi_i \equiv (\exists L_2) \psi_j$.

This *consistency condition* will be maintained in our analysis. However, maintaining this condition through the abstract interpretation process is tricky.

Example 2. Consider a simple action by a thread T_1, $a : \ell_1 \xrightarrow{x \; := \; 10} \ell_2$ which updates a shared variable x to 10, while thread T_2 remains in location m. This corresponds to a move in the transaction graph from $\langle \ell_1, m \rangle \to \langle \ell_2, m \rangle$.

Let $\langle (x = 5), (x = l, l = 5) \rangle$ be the assertion labeling the node (ℓ_1, m). After a move by the first thread, the assertion labeling the node (ℓ_2, m) should be $\langle (x = 10), (l = 5, x = 10) \rangle$. Note that to maintain consistency, we are forced to update the invariant for thread T_2 even if T_2 did not perform any action.

A similar situation arises at the "join-nodes" of the transaction graph. Due to the consistency condition, these nodes do not act as true join nodes as in the sequential case. We address these difficulties by introducing a "meld" operator in the abstract domain, in order to maintain the consistency condition.

Melding: As mentioned earlier, the shared variables S may be modified by executing some transaction edge $n_i \to n_k$ by thread T_i, updating the component φ_i of the invariant tuple. However, this may violate the consistency requirements w.r.t the invariant tuples corresponding to the other threads. To enforce this consistency, we introduce an operator $\mathsf{meld}(\varphi_k, \psi_j)$ that forces the global state values represented by assertion ψ_j to coincide with those in φ_k.

Definition 2 (Meld). *Let $\alpha[S, L_1]$ and $\beta[S, L_2]$ be assertions over shared and local variables for each thread. The assertion $\gamma : \mathsf{meld}(\alpha, \beta)$ is such that (a) $(\exists L_1)\alpha \equiv (\exists L_2)\gamma$ and $(\exists G)\beta \models (\exists G)\gamma$, and (b) the operator must be entry-wise monotonic. I.e., if $\alpha_1 \models \alpha_2$ and $\beta_1 \models \beta_2$ then*

$$\mathsf{meld}(\alpha_1, \beta_1) \models \mathsf{meld}(\alpha_2, \beta_1), \quad \mathsf{meld}(\alpha_1, \beta_1) \models \mathsf{meld}(\alpha_1, \beta_2).$$

The result γ of $\mathsf{meld}(\alpha, \beta)$ over-approximates the global variable values described by α and the local variable values described by β.

The design of a melding operator is specific to the underlying abstract domain. A simple melding operator can be constructed for most abstract domains using projection and works for domains wherein the conjunction $\sqcap$ coincides with the logical conjunction $\wedge$ (i.e, Moore closed domains). Formally, we have $\mathsf{meld}(\varphi, \psi)$: $(\exists L_1)\varphi \wedge (\exists G)\psi$.

Post Condition: An elementary step in the fixed point computation consists of propagating an assertion pair $\langle \varphi_i, \psi_j \rangle$ across an edge $n_i \rightarrow n_k$ of one of the threads. Let φ_k denote the result of the post-condition $post(\varphi_i, n_i \rightarrow n_k)$. In practice, however, a move by a thread $n_i \rightarrow n_k$ in the transaction graph may represent the execution of a (possibly atomic) program segment in the corresponding thread consisting of numerous basic blocks. Therefore, the "post" needs to be computed using a thread-local abstract interpretation of the segment corresponding to the edge.

The effect of executing a thread edge $n_i \rightarrow n_k$ starting from the node $\langle n_i, m_j \rangle$, labelled by the assertion $\langle \varphi_i, \psi_j \rangle$, yields the assertion pair $\langle \varphi_k, \psi_j' \rangle$ wherein: φ_k : $post(\varphi_i, n_i \rightarrow n_k)$, ψ_j' : $\mathsf{meld}(\varphi_k, \psi_j)$. Formally, we use a propagation operator $\mathsf{propagate}$ to model the effect of executing a transaction n_i across an edge $n_i \rightarrow n_k$: $\mathsf{propagate}(\langle \varphi_i, \psi_j \rangle, n_i \rightarrow n_k) = \langle \varphi_k, \psi_j' \rangle$.

Our goal is to produce a map η labeling each node of the transaction graph $\langle n_i, m_j \rangle$ with a pair of assertions $\eta(n_i, m_j)$: φ_i, ψ_j such that $\varphi_i[S, L_1]$ relates the shared variables S with the local variables L_1, and similarly ψ_j. Secondly, each of the assertions $\langle \varphi_i, \psi_j \rangle$ holds, whenever the individual thread controls simultaneously reside in the nodes $\langle n_i, m_j \rangle$.

Formally, for any edge $\langle n_i, m_j \rangle \rightarrow \langle n_k, m_j \rangle$, we require that $\mathsf{propagate}(\eta(n_i, m_j), n_i \rightarrow n_k) \models \eta(n_k, m_j)$. A symmetric condition needs to hold for moves of the thread T_2. The map η can be constructed using forward propagation on a transaction graph G using $\mathsf{propagate}$ as the post-condition.

Loops and Recurrences: A cycle in the transaction graph corresponds directly to a loop or a recursive procedure in one or more of the threads. Such cycles are handled naturally in our abstract interpretation scheme using widening. Specifically, widening is performed conservatively at each node of the form $(l_1, \ldots, l_k)$ such that for some component l_i there exists a back-edge of the form $m_i \rightarrow l_i$ in the CFG of T_i. For example, in the case of cycles arising due to loops l_i would be a loop head. Standard iteration schemes known for sequential programs can be used for analyzing transaction graphs.

5 Refinement of Transaction Graphs

We use the abstract interpretation framework described in the previous section to automatically derive sound invariants for the concurrent program. In practice, we use abstract domains of increasing precision ranges, octagons, and polyheda to derive more accurate invariants.

Algorithm 2. Refinement of Transaction Graph

1: Construct an initial transaction graph G_π^0 by using partial order techniques and
   synchronization constraints.
2: **repeat**
3:    Compute range, octagonal, and polyhedral invariants over G_π^i And prune paths
      from G_π^i resulting in H_π^i.
4:    Compute a new product transaction graph G_π^{i+1} based on the thread conflicts
      and synchronization constraints in H_π^i.
5: **until** $G_\pi^{i+1} = G_\pi^i$
6: **return** G_π^i

Invariant-based slicing of thread conflicts. At each stage, we use the de-
rived invariants to show the unreachability of code segments, e.g. in conditional
branches. If these unreachable code segments contain shared variable accesses,
this can lead to a reduction in the conflicts between threads, thereby allow-
ing larger transactions in individual threads. We call this a *refinement* of the
transaction graph, since it provides a more accurate view of thread interleavings
required for analysis. Such refinement helps to improve accuracy of subsequent
analysis by discounting spurious thread interference from unreachable code seg-
ments, while also improving scalability due to smaller transaction graphs that
result from a smaller number of interleavings and larger individual transactions.

Iterative refinement. In general, we can iteratively refine the transaction
graph by alternately leveraging conflict analysis (using partial order techniques
and synchronization constraints) and sound invariants until we reach a fix-point,
where the transaction graph cannot be refined any more.

This iterative procedure for refining a transaction graph is shown in Figure 2.
The initial transaction graph construction utilizes POR and synchronization con-
straints (described in Section 3). This *bootstraps* the iterative process. This initial
step is critical for making the computation of sound invariants scale (described
in Section 4). This is because the initial transaction graph over global control
states is much smaller than a naive product graph over individual statements
in threads. Furthermore, the capturing of POR and synchronization constraints
drastically reduces the number of interleavings considered by our invariant com-
putation. This effectively, makes the invariants stronger.

6 Applications

The transaction graph constructed by exploiting synchronization constraints
and sound invariants can be used for various analyses and verification appli-
cations on concurrent programs. These include concurrent pointer alias analysis,
model checking, etc. Once the product transaction graph has been computed,
any dataflow analysis of concurrent programs can be carried out sequentially
over the nodes of this graph. From a model checking perspective, the product

transaction graph encodes all the context switches that need be explored. When carrying out partial order reduction over the state space during model checking (described later in this section), we allow context switching only at transaction boundaries defined by the transaction graph.

In the remainder of this section, we describe a specific application of our approach for detection of data race bugs in concurrent programs. There have been many successful efforts based on static analysis [28,25,24,11,26,15]. These approaches, however, may generate a large number of bogus warnings. Model checking [5,3,6] has the advantage that it can produce concrete error traces and does not rely on the programmer to inspect the warnings and decide whether they are true bugs or not. However, the state explosion problem severely limits its scalability, especially on large real-life concurrent programs.

Classic static data race warning generation has three main steps. First, control locations with shared variable accesses are determined in each thread. Next, the set of locks held at each of these locations of interest are computed, using lockset analysis. Pairs of control locations in different threads where (i) the same shared variable is accessed, (ii) at least one of the accesses is a write operation, and (iii) disjoint locksets are held, constitute a potential data race site and a warning is issued.

Since dataflow analysis for concurrent programs is undecidable in general, typical static data race detection methods ignore conditional statements in the threads and perform thread-local analysis only. Indeed, a pair of control locations (c_1, c_2) marked as a potential data race site may simply be unreachable in any run of the given concurrent program.

We use the static analysis framework proposed in this paper to check the reachability of the pair of control locations (c_1, c_2) appearing in such warnings. If the pair is statically unreachable, then the warning is bogus, and can be eliminated. The combined use of synchronization constraints and sound invariants provide cheaper methods than model checking to check the pairwise (un)reachability of c_1 and c_2, while providing more accuracy than existing static analysis methods for data race detection. In fact, one can use any of the existing fast methods to generate the initial set of data race warnings, and use our techniques to automatically reduce the number of warnings.

We also leverage the final transaction graph generated in our framework to perform model checking, for producing concrete error traces for the remaining warnings. Details of our symbolic (SAT-based) model checking techniques for concurrent programs are described in our previous work [19]. The additional benefit is that our transaction graph already captures reductions in thread interleavings that would have otherwise been explored during model checking. We also use slicing on the transaction graph to generate smaller models for specific warnings, by inlining the functions in the specific contexts and slicing away the rest. We can also use the derived invariants to prune the search space during model checking. The combined effect is to improve the viability of model checking on concurrent programs.

7   Experimental Results

We applied the proposed static analysis framework for reducing data race warnings generated by an initial lockset-based analysis on a suite of Linux device drivers with known data races. The results are shown in Table 1, where columns 4 and 5 report the number of warnings (#Warn) and time taken (seconds), respectively, by the lockset-based static analysis. Column 6 reports the number of warnings after reduction by using our invariant-based static analysis, with the time taken (in seconds) reported in Column 7. This analysis is successful in reducing the number of warnings to a more manageable level within a few minutes. As an additional benefit, we may now apply techniques such as model checking on the few remaining warnings. Column 8 reports the number of warnings for which our model checking procedure [19] was successful in generating a concrete error trace, with the final unresolved number of warnings reported in Column 9.

Table 1. Results for Static Reduction of Data Race Warnings

Driver	KLOC	# $ShVars$	#Warn.	Time (secs)	#After Invar	Time (secs)	#Wit.	#Unknown
pci_gart	0.6	1	1	1	1	4	0	1
jfs_dmap	0.9	6	13	2	1	52	1	0
hugetlb	1.2	5	1	3.2	1	0.9	1	0
ctrace	1.4	19	58	6.7	3	143	3	0
autofs_expire	8.3	7	3	6	2	12	2	0
ptrace	15.4	3	1	15	1	2	1	0
raid	17.2	6	13	1.5	6	75	6	0
tty_io	17.8	1	3	4	3	11	3	0
ipoib_multicast	26.1	10	6	7	6	16	4	2
TOTAL			99		24		21	3

Table 2. Results for Model Checking Data Race Warnings. All timings are in seconds and memory in MBs.

Witness #	Symbolic POR+BMC			Witness #	Symbolic POR+BMC		
	Depth	Time	Mem		Depth	Time	Mem
jfs_dmap : 1	10	0.02	59	raid : 4	34	4.15	61
ctrace : 1	8	2	62	raid : 5	40	9.30	59
ctrace : 2	56	10 hr	1.2G	raid : 6	70	70	116
ctrace : 3	92	2303	733	tty_io : 1	34	0.82	5.7
autofs_expire : 1	9	1.14	60	tty_io : 2	32	9.69	14
autofs_expire : 2	29	128	144	tty_io : 3	26	31	26
ptrace : 1	111	844	249	ipoib_multicast : 1	6	0.1	58
raid : 1	42	26.13	75	ipoib_multicast : 2	8	0.1	59
raid : 2	84	179	156	ipoib_multicast : 3	4	0.1	58
raid : 3	44	32.19	87	ipoib_multicast : 4	14	0.3	59

The detailed results for model checking are shown in Table 2, where we report the depth at which the bug is found, and the time and memory used by our model checking procedure that uses symbolic POR with SAT-based BMC. We were able to generate a concrete error trace for the known data race in all but one example. This is mainly due to the small sliced models we generated by using warning-specific static information, even for large drivers (such as *ipoib_multicast*). Thus, our static analysis framework enables scalable model checking for larger concurrent programs.

8   Related Work and Conclusions

We have presented a general framework for static analysis of concurrent programs, where we use partial order reduction and synchronization constraints to capture a reduced set of thread interleavings, on which we derive sound invariants by using abstract interpretation to perform further reduction. We described an application of this framework to reduce the number of data race warnings, and to enable the application of model checking to find concrete error traces.

Our work is related to prior work on verification of concurrent programs that attempts to get around the undecidability barrier by considering restricted models of synchronization and communication [1,12] or by bounding the number of context switches [27,2,23,21]. There are also other recent efforts to leverage sequential analysis in concurrent settings [4,20]. Our approach also exploits specific patterns of synchronization, but our main focus is on deriving *sound* invariants for reduction in thread interleavings, by lifting abstract interpretation techniques to the concurrency setting. Since thread interleavings are a primary source of complexity in concurrent programs, this provides us further opportunities to apply more precise analyses, including model checking.

References

1. Atig, M.F., Bouajjani, A., Touili, T.: On the reachability analysis of acyclic networks of pushdown systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 356–371. Springer, Heidelberg (2008)
2. Bouajjani, A., Fratani, S., Qadeer, S.: Context-bounded analysis of multithreaded programs with dynamic linked structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
3. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: ASE (2000)
4. Chugh, R., Voung, J.W., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: PLDI, pp. 316–326 (2008)
5. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logics of Programs, pp. 52–71 (1981)
6. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE (2000)
7. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming (1976)
8. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to Abstract interpretation, invited paper. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)

9. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: ACM POPL, January 1978, pp. 84–97 (1978)
11. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: SOSP (2003)
12. Farzan, A., Madhusudan, P.: Causal dataflow analysis for concurrent programs. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 102–116. Springer, Heidelberg (2007)
13. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
14. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: F-SOFT: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
15. Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: PLDI (2002)
16. Kahlon, V.: Bootstrapping: A Technique for Scalable Flow and Context- Sensitive Pointer Alias Analysis. In: PLDI (2008)
17. 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)
18. Kahlon, V.: Exploiting Program Structure for Tractable Dataflow Analysis of Concurrent Programs (2008), `kahlonnec-labs.com`
19. Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 286–299. Springer, Heidelberg (2006)
20. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
21. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
22. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
23. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: PLDI (2008)
24. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL (2007)
25. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI (2006)
26. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In: PLDI (2006)
27. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
28. Sterling, N.: Warlock: A static data race analysis tool. In: USENIX Winter Technical Conference (1993)

Inferring Synchronization under Limited Observability

Martin Vechev, Eran Yahav, and Greta Yorsh

IBM T.J. Watson Research Center

Abstract. This paper addresses the problem of automatically inferring synchronization for concurrent programs. Given a program and a specification, we infer synchronization that avoids all interleavings violating the specification, but permits as many valid interleavings as possible. We let the user specify an upper bound on the cost of synchronization, which may limit the *observability —* what observations on program state can be made by the synchronization code. We present an algorithm that infers, under certain conditions, the *maximally permissive* synchronization for a given cost. We implemented a prototype of our approach and applied it to infer synchronization in a number of small programs.

1 Introduction

Concurrency is hard. Concurrent execution of operations that share data requires synchronization to guarantee correctness. Typically, the programmer is required to reason about all the ways in which concurrent operations can interleave, and introduce synchronization code that avoids incorrect interleavings. Because of the excruciating difficulty in finding even a single choice of synchronization that makes the program correct and reasonably efficient [15], programmers often introduce synchronization in an ad-hoc manner, and rarely explore alternative choices. In particular, programmers often resort to coarse-grained synchronization because: (i) it simplifies reasoning about the program, and (ii) the overhead incurred by finer-grained synchronization is prohibitive.

Our goal is to assist the programmer in systematically exploring alternative choices of synchronization, based on the cost that she is willing to accept. Given a program P, and a specification S, we define the set $VP(P, S)$ of concurrent programs that satisfy S and can be obtained from P by adding synchronization. To understand the tradeoffs between different choices of synchronization code, we examine two dimensions along which programs in $VP(P, S)$ can be compared:

- **Permissiveness:** Given two programs $P_1, P_2 \in VP(P, S)$, we say that P_1 is *more permissive* than P_2 when the set of traces permitted by P_1 is a superset of the set of traces permitted by P_2.
- **Synchronization Cost:** Given two programs $P_1, P_2 \in VP(P, S)$, we say that P_1 has *lower cost* than P_2 when the running time of the synchronization code in P_1 is lower than that of P_2.

There is a connection between the cost of synchronization and its permissiveness. For the synchronization code to be more permissive, it needs to draw finer distinctions between interleavings, which typically requires atomically observing more of the program's state. Atomically observing more of the program's state means increasing the synchronization cost.

In general, the user would like to maximize permissiveness and minimize the cost. However, the synchronization solution that provides maximal permissiveness maybe too costly. There may be another (incomparable) solution, with less permissiveness and lower cost, which is acceptable. We let the user specify an upper bound on the cost, and infer a maximally permissive solution within the limits of this upper bound.

There are various synchronization mechanisms available to concurrent programmers today. In this paper we choose to focus on the classical conditional critical regions (CCRs), an elegant construct originally introduced by Hoare [7]. A CCR consists of a guard and a sequence of statements that are to be executed atomically if the guard evaluates to *true*. If the guard evaluates to *false*, the thread blocks until it is able to atomically re-evaluate the guard. Guards only observe program state, but cannot modify it. CCRs have been implemented as a synchronization construct in the language Edison [5], as a language extension of Java via software transactional memory [6], and recently in the high-performance parallel language X10 [14]. One of the advantages of CCRs over other lower-level operational primitives such as locks and condition variables is their concise and declarative nature.

A key challenge in using CCRs is finding the appropriate guard expressions. A programmer must address the following: (i) correctness — guards must eliminate invalid interleavings that violate S; (ii) permissiveness — guards should allow as many interleavings as possible: a thread executing a guard should not block unless its execution is doomed to violate S; (iii) cost — it is important to reduce the cost of evaluating the guard expression. Because the guard is evaluated atomically, this cost is typically dictated by the number of shared variables accessed in the guard. One way to reduce the cost is by restricting the code to observe only a subset of these variables. Balancing these trade-offs may require the programmer to simultaneously consider all guards in all of the CCRs in the program.

This work addresses the challenge of automatically inferring correct and maximally permissive guards, without exceeding the upper bound on the cost of the guards, specified by the user. This bound restricts the *language of guards* — the expressions that can be used as guards — to those that cost less than the specified bound.

Consider a concurrent program P, a specification S, and a language of guards LG. We denote by $VP(P, S, LG)$ the set of programs that satisfy S and are obtained from P by adding guards from LG. It is possible that no program $P' \in VP(P, S, LG)$ permits all valid interleavings of P. The reason is that the language LG may not be expressive enough to distinguish between a valid and an invalid interleaving and thus a valid program P' must avoid both. It is therefore natural to define the notion of a *maximally-permissive program* under a given language of guards: $P' \in VP(P, S, LG)$ is maximally-permissive with respect to LG if there is no program in $VP(P, S, LG)$ that permits more interleavings than P'. In other words, it is impossible to modify P' using expressions from LG to permit more interleavings without violating the specification S. Our goal in this paper can be stated as follows:

> Given a concurrent program P, a specification S, and a language of guards LG, construct a program $P' \in VP(P, S, LG)$, such that P' is maximally permissive with respect to LG, and has minimal synchronization cost.

The above problem statement is closely related to the ones addressed by program repair [9] and controller synthesis [12]. However, in contrast to these approaches, our work focuses on inferring synchronization code that observes the state without modifying it, and takes into account the cost of synchronization when attempting to find the maximally permissive solution.

1.1 Main Contributions

The contributions of this paper can be summarized as follows:

- We present a technique for automatically inferring correctly-synchronized concurrent programs. To explore alternative choices of synchronization, we let the user control the upper bound on the cost.
- We first present an exponential algorithm that infers a maximally permissive program for a given language of guards. Next, we define a greedy algorithm that infers, under certain conditions, a maximally permissive program for the given language of guards. Both algorithms minimize synchronization cost.
- We implemented a prototype of our approach and applied it to several programs, including classical ones such as dining philosophers and asynchronous counters.

Next, we use a simple example to illustrate the challenges that our goal presents, and show how they are addressed in our approach.

1.2 A Simple Motivating Example

Fig. 1 is a simple program consisting of three operations op1, op2, and op3, that are executed concurrently by the client program (the main procedure). The interleavings for this example are shown in Fig. 2. In this example, the global state consists of the program counter of each of the three threads, and the value of the shared variables x, y, z. We denote the global state using a tuple $\langle pc_1, pc_2, pc_3, x, y, z \rangle$ where pc_1, pc_2, pc_3 are program counters and x, y, z are the values of the corresponding shared variables. For this program, we would like to guarantee that the global invariant $y \neq 2 \lor z \neq 1$ is maintained. Unfortunately, while most interleavings indeed satisfy this specification, the interleaving x=z+1;z=y+1;y=x+1 leads to its violation. In the figure, we use nodes with red dotted boundaries to denote states in which the invariant is violated.

Implementability. Our goal in the example is to construct a new maximally permissive program in which the invalid interleaving above is not allowed. Generally, to eliminate invalid traces, we consider the (possibly infinite) set of program traces represented using a transition system, and compute a subset of the transitions in the transition system for which all resulting traces are guaranteed to be accepting. However, since our goal is to

```
op1 { 1: x = z + 1 }          main:
op2 { 2: y = x + 1 }              int x = 0, y = 0, z = 0;
op3 { 3: z = y + 1 }              op1 || op2 || op3
```

Fig. 1. An example program with three threads

construct a program, it is not sufficient to find a valid transition system, we need to find one that is expressible as a program in the provided programming language. Similar *implementability* challenges occur in other synthesis settings, e.g., synthesis of reactive modules [12].

Cost vs. Permissiveness. The ability to avoid a specific transition depends on the amount of information that can be obtained atomically from the global state and reflected in a CCR guard. Atomically reading the entire program state is often too costly. Reducing the cost of synchronization is achieved by restricting the language of guards. When the language of guards is restricted, the information available for a guard might not be sufficient to uniquely identify a single transition. This *limited observability* induces a natural equivalence between transitions. Informally, we define two transitions to be equivalent when they execute the same statement, and their source states cannot be distinguished by the language of guards. Under limited observability, the addition of a guard to a statement in order to eliminate a transition t results in the elimination of *all transitions that are equivalent to* t.

Fig. 3(a) shows a valid version of the example of Fig. 1 using CCRs with guards where the bound on the cost allows the solution to observe the entire program state. The synchronization in this program was automatically inferred by our tool. In this program, the guard $(x \neq 1 \lor y \neq 0 \lor z \neq 0)$ (directly) eliminates *only* the transition $\langle e, 2, 3, 1, 0, 0 \rangle \stackrel{z=y+1}{\longrightarrow} \langle e, 2, e, 1, 0, 1 \rangle$ which would have inevitably led to an error state. Note that in this example, allowing the guards to observe the values of all shared variables leads to the maximally permissive result of only eliminating invalid interleavings.
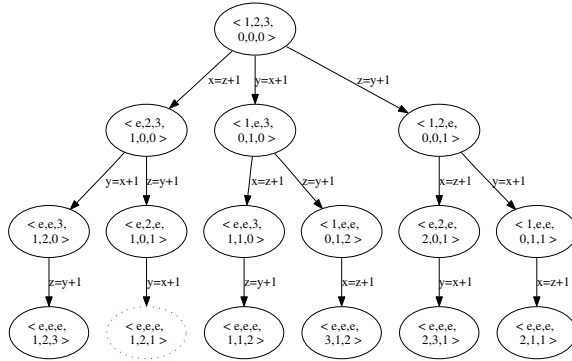
Fig. 2. Transition system for the example program of Fig. 1 (Self-loops on exit states are omitted.)

However, suppose that our solution is restricted to use CCR guards whose cost is limited to only observing the values of variables x, z (and not the entire program state). Under such limited observability, the states $\langle e, 2, 3, 1, 0, 0 \rangle$, $\langle e, e, 3, 1, 2, 0 \rangle$, and $\langle e, e, 3, 1, 1, 0 \rangle$ cannot be distinguished by any guard. Therefore, the guard $(x \neq 1 \lor z \neq 0)$ added to the statement z=y+1 to eliminate the bad transition $\langle e, 2, 3, 1, 0, 0 \rangle \stackrel{z=y+1}{\longrightarrow} \langle e, 2, e, 1, 0, 1 \rangle$ has the side effect of eliminating the two other equivalent transitions. This triggers further elimination of transitions from state $\langle 1, e, 3, 0, 1, 0 \rangle$ of statement x=z+1. Fig. 3(b) shows a valid solution of the example of Fig. 1 inferred by our tool.

```
op1 { 1: x = z + 1 }                op1 { 1: (x ≠ 0 ∨ z ≠ 0) ->
op2 { 2: y = x + 1 }                       x = z + 1 }
op3 { 3: (x ≠ 1 ∨ y ≠ 0 ∨ z ≠ 0) -> op2 { 2: y = x + 1 }
        z = y + 1 }                 op3 { 3: (x ≠ 1 ∨ z ≠ 0) ->
                                           z = y + 1 }
            (a)                                (b)
```

Fig. 3. Example program with synchronization, observing (a) all shared variables , (b) only x, z

Sometimes it is possible to simplify the guards of a solution *without* affecting the set of allowed interleavings. For example, in Fig. 3(a), we can use only variables x and y in the guard of z=y+1. Such optimizations are further discussed in Section 4.1.

The key point to take away from this example is the connection between synchronization cost and permissibility. Restricting the cost of synchronization by limiting observability may lead to eliminating valid interleavings that cannot be distinguished from invalid ones. For instance, the solution in Fig. 3(b) permits a subset of the traces allowed by the solution in Fig. 3(a) because it is not allowed to observe variable y.

In the rest of the paper we describe our approach in more detail. Due to space restrictions, our description is somewhat informal. Additional formal details, examples, proofs, and discussion of related work are available in [16].

2   Preliminaries

Transition System. A transition system ts is a tuple $\langle \Sigma, T, Init \rangle$ where Σ is a set of states, $T \subseteq \Sigma \times \Sigma$ is a set of transitions between states, and $Init \subseteq \Sigma$ are the initial states. For a transition $t \in T$, we use $src(t)$ to denote the source state of t, and $dst(t)$ to denote its destination state.

For a transition system ts, we use the following notations. We use $s \rightsquigarrow_{ts} s'$ to denote that there exists a path in ts starting in state s and ending in state s'. Formally, the relation $\rightsquigarrow_{ts}$ is the reflexive transitive closure of the successor relation defined by T. A stuck state is a state that does not have any successors in ts. The set of stuck states is denoted by $Stuck_{ts}$. A doomed state is a state from which all paths end in stuck states. The set of doomed states is denoted by $Doomed_{ts}$. We say that a state $s \in \Sigma$ is reachable when there exists a path to s from some initial state. The set of all reachable states of ts is denoted by $Reach_{ts}$. A transition system ts is valid, denoted by $valid(ts)$, if and only if no doomed state is reachable. For a transition system ts, a trace is a (possibly infinite) sequence of transitions $t_0, t_1, \ldots$ such that $src(t_0) \in Init$ and for every $i \geq 0$, $t_i \in T$ and $dst(t_i) = src(t_{i+1})$. We use $[\![ts]\!]$ to denote the set of traces of a transition system ts. A trace is valid if it does not contain any doomed state.

Conditional Critical Regions (CCRs). The conditional critical region (CCR) construct, originally introduced by Hoare [7], allows the programmer to specify what operations have to be executed atomically and under what condition. A CCR has the form: $guard \rightarrow stmt$ where $guard$ is a boolean expression and $stmt$ is a statement (including a sequential composition of statements) that have to be executed atomically. The guard is evaluated atomically and if *true*, the statements are executed atomically. Otherwise, the thread blocks until the guard evaluates to *true*.

Program Syntax. For the purpose of this paper, we consider a program that consists of a set of (named) operations, $Op \stackrel{\text{def}}{=} \{op_1, \ldots, op_n\}$, executed in parallel by different threads. An operation is a code fragment defined using a simple, flat, programming language with assignment, conditional and unconditional goto, sequential composition, and CCRs. The language does not contain parallel composition, allocation of threads, nested CCRs, and invocation of operations.

If not stated otherwise, each basic statement is in a separate CCR, guarded by $true$, and the guard is omitted. The user may define CCRs in which the atomic statement consists of a sequence of statements, and not a single basic statement. We assume that every statement participates in (exactly one) CCR.

We use *Var* to denote the set of (shared) program variables, which can be referenced by any operation. To simplify the exposition, we do not use local variables. There is nothing in our approach that prevents us from using local variables, but having local variables makes the formal definitions cumbersome. We assume that all program variables have integer values, initialized to 0.

Program Semantics. Let P be a program with variables *Var*. A program state s is a pair $\langle pc_s, val_s \rangle$ where $pc_s \colon \{1, \ldots, n\} \to$ *Int* maps a thread identifier to the program counter of the corresponding thread, ranging over program locations in the operation executed by the thread, and $val_s \colon$ *Var* $\to$ *Int* is a valuation of the variables. We use Σ_P to denote the set of all program states. The set of initial program states is denoted by $Init_P \subseteq \Sigma_P$. The value of a program expression e in a state $s \in \Sigma_P$ is denoted by $[\![e]\!](s)$. It is computed using standard evaluation rules for program expressions.

We define a transition system for a program P to be $\langle \Sigma_P, T_P, Init_P \rangle$ where a transition $(s, s') \in T_P$ is labeled by a program location l and a thread identifier tid. A transition (s, s') labeled with l and tid is in T_P if (i) the program counter of the thread tid in state s is at program location l, (ii) the guard of the CCR at program location l is satisfied in s, and (iii) execution of the statement corresponding to CCR at l in program state s by thread tid results in state s'. In addition, we guarantee that states at the program exit are not stuck by adding the corresponding self-loop transitions to ts. For a transition $t \in T_P$, we use $lbl(t)$, $tid(t)$, and $ccr(t)$ to denote the corresponding program location, thread id, and the (unique) CCR at program location $lbl(t)$, respectively.

The semantics of a program P, denoted by $[\![P]\!]$, is the (prefix closed) set of traces of the corresponding transition system $\langle \Sigma_P, T_P, Init_P \rangle$.

Specification. The user can specify a global invariant S, which describes a set of states. An invariant can refer to program variables and to the program counter of each thread (e.g., to model local assertions). Our approach can be extended to handle any temporal safety specifications, expressed as a property automaton, by computing the synchronous product of program's transition system and the property automaton [3].

We define $\langle \Sigma_P, T_{P,S}, Init_P \rangle$ to be a transition system for a program P and global invariant S, where $T_{P,S} \subseteq T_P$ is defined by removing from T_P all transitions in which the source state does not satisfy S: $T_{P,S} = \{t \in T_P | src(t) \text{ satisfies } S\}$. This effectively means that in the transition system for P and S, all states which do not satisfy S become stuck states — states with no outgoing transitions. If a stuck state is reachable, the transition system for P and S is not valid.

A program P is valid with respect to S if and only if the corresponding transition system $\langle \Sigma_P, T_{P,S}, \mathit{Init}_P \rangle$ is valid. This notion of validity includes both safety properties defined by the global invariant S and a progress guarantee that the program does not get stuck, in any execution.

3   Maximally-Permissive Programs

Given an input program P and a specification S, we modify P by adding synchronization such that the modified program satisfies the specification S. Conceptually, we take the following steps: (i) construct the transition system ts of P and S; (ii) remove a minimal set of transitions from ts such that the resulting transition system ts' is valid with respect to S; (iii) implement ts' as a program, by adding synchronization code to P.

In this work, we rely on standard techniques to construct the transition system of P, e.g., [8], and focus on steps (ii) and (iii).

3.1   Removing Transitions under Limited Observability

By limiting the cost of synchronization code, we induce limited observability. Hence, not every transition system obtained by removing a bunch of transitions from ts can be implemented as a program with the same traces by adding synchronization code to P.

To remove a transition t, and implement the result as a program, the input program P is modified by strengthening the guard of $ccr(t)$, preventing its execution from the source state $src(t)$. When the state $src(t)$ can be uniquely characterized by an expression in the language of guards LG, we can use its characterization to strengthen the guard of $ccr(t)$ without affecting transitions other than t. Our ability to uniquely characterize a state $src(t)$ depends on LG. Usually, due to limited observability, we may not be able to uniquely characterize $src(t)$. In such cases, the removal of the transition t may remove other transitions executing the same statement, because they have the same guard. We say that two transitions are *equivalent* when the language of guards is not expressive enough to remove one of the transition without removing the other one. We now provide a formal definition of the transition equivalence under limited observability.

Observational Equivalence.   First, we define equivalence relation on states with respect to LG. Two states are equivalent with respect to LG, when there is no guard in LG that can be used to distinguish them. Formally, for all $s, s' \in \Sigma_P$,

$$s \approx_{LG} s' \text{ if and only if for all } g \in LG. [\![g]\!](s) = [\![g]\!](s') \tag{1}$$

We now define equivalence relation on transitions with respect to LG. Two transitions t and t' are equivalent when they execute the same statement and their source states are indistinguishable by LG. Formally, for all $t, t' \in T_{P,S}$,

$$t \approx_{LG} t' \text{ if and only if } lbl(t) = lbl(t') \text{ and } src(t) \approx_{LG} src(t') \tag{2}$$

We use $[t]_{LG}$ to denote the equivalence class of t with respect to $\approx_{LG}$. For a set of transitions $E \subseteq T_{P,S}$, we use $[E]_{LG}$ to denote $\cup_{t \in E}[t]_{LG}$.

Characterizing Observable States. We define a characterization function to respect the equivalence relation $\approx_{LG}$. Let χ be a function that takes as input a state $s \in \Sigma_P$ and returns a guard in *LG*. We say that χ characterizes the states observable by *LG*, when for all $s, s' \in \Sigma_P$,

$$[\![\chi(s)]\!](s') = true \text{ if and only if } s \approx_{LG} s' \qquad (3)$$

Our method is applicable to any guard languages for which a characterization function is defined. Usually, it is easy to define a characterization function, e.g., by enumerating the values of observable variables in the state.

Example 1. Consider the program of Fig. 1 and its transition system in Fig. 2. Let *LG* be boolean combinations of predicates of the form $var == c$, where var is one of the program variables $\{x, z\}$, and c is a constant. Under *LG*, many of the states in Fig. 2 are equivalent. For example, the states $s_1 = \langle e, 2, 3, 1, 0, 0 \rangle$, $s_2 = \langle e, e, 3, 1, 2, 0 \rangle$, and $s_3 = \langle e, e, 3, 1, 1, 0 \rangle$ are equivalent as they cannot be distinguished by *LG*. Consequently, the transitions corresponding to the statement z=y+1 outgoing from s_1, s_2, and s_3 are equivalent. When the characterization function is defined by enumerating the values of observable variables in the state, $\chi(s_1) = \chi(s_2) = \chi(s_3) = (x == 1) \wedge (z == 0)$.

3.2 Implementability

We can use χ to define a guard in *LG* that removes a transition $t \in T_{P,S}$, and all the transitions in its equivalence class $[t]_{LG}$, but does not affect any other transitions.

Lemma 1. *For all $t, t' \in T_{P,S}$ such that $lbl(t) = lbl(t')$, $t' \approx_{LG} t$ if and only if $[\![\chi(src(t))]\!](src(t')) = true$.*

A transition system ts is implementable from P and *LG* when there exists a program P' obtained from P by introducing guards from *LG* such that the set of traces of ts and P' are the same. The following theorem relates implementability to observational equivalence. Intuitively, if we remove an equivalence class of transitions from an implementable transition system, the result is an implementable transition system.

Theorem 1 (Implementability). *For every $R \subseteq T_{P,S}$, the transition system ts defined by $\langle \Sigma_P, T_{P,S} \setminus [R]_{LG}, Init_P \rangle$ is implementable from P and LG:*
(1) There exists a program P' such that $[\![P']\!] = [\![ts]\!]$.
(2) P' can obtained from P by introducing guards from LG.

Given P and $[R]_{LG}$, for some $R \subseteq T_{P,S}$, the simple algorithm `implement` from Fig. 4 computes such P'. It relies on Lemma 1 to guarantee that only transitions from $[R]_{LG}$ are removed. The algorithm constructs P' from P by strengthening the guards of CCRs that corre-

```
implement(P:Program,R:Transitions):Program {
  P' = P
  foreach t ∈ R
    let ccr(t) be l : guard → stmt in
    P' = P'[l : ¬χ(src(t)) ∧ guard → stmt]
  return P'
}
```

Fig. 4. The procedure `implement`

spond to transitions in $[R]_{LG}$. For a transition t, we use the notation $P'[l : \neg\chi(src(t)) \wedge guard \rightarrow stmt]$ for the program obtained from P' by strengthening the guard of $ccr(t)$ to be $\neg\chi(src(t)) \wedge guard$. This change is sufficient (by Lemma 1) to remove the transition t itself and all its equivalence class $[t]_{LG}$, but only them.

3.3   Maximally Permissive Programs

We now define the natural notion of a maximally-permissive program for a given language of guards. We note that maximal permissiveness arises in many other settings (e.g., [10,13]).

Definition 1 (Maximally-Permissive). *Consider a program P and a language of guards LG. Let P' be a program obtained from P by introducing guards from LG. P' is maximally-permissive with respect to LG if and only if P' is valid and for every program P'' obtained from P by introducing guards from LG, if $[\![P']\!] \subset [\![P'']\!]$, then P'' is not valid.*

We use $MP(P, LG)$ to denote the set of all maximally-permissive programs that can be obtained from P by introducing guards from LG. Note that the programs in $MP(P, LG)$ have identical or incomparable sets of traces, i.e., for every pair $P, P' \in MP(P, LG)$, $[\![P]\!] \not\subset [\![P']\!]$. When we cannot eliminate all invalid interleavings (that end in stuck states) only by introducing guards, $MP(P, LG)$ is empty.

In the rest of this section, we show that every maximally-permissive program can be implemented by removing edges from the transition system of P.

We present two algorithms for computing maximally permissive programs with respect to the language of guards *LG*. The language *LG* is required in all of the algorithms. To avoid clutter we do not pass it as an explicit parameter.

3.4   EXHAUSTIVE Algorithm

Theorem 1 allows us to implement any transition system defined by removing a set of transitions $[R]_{LG}$ from the transition system that corresponds to the original program P. We are interested in valid transition systems. Therefore, we restrict our attention to sets of transitions that yield *valid and implementable* transition systems. Rather than considering all subsets of transitions as possible candidates for removal, we define the set of *bad transitions*, and only consider transitions from this set as candidates for removal.

We define a bad transition as a transition that lies on an invalid trace. More formally, given a transition system $\langle \Sigma, T, Init \rangle$ we say that a transition $t \in T$ is a *bad transition* when $i \rightsquigarrow_{ts} src(t), dst(t) \rightsquigarrow_{ts} d$, such that $i \in Init, d \in Doomed_{ts}$. Using this definition, we would like to construct an algorithm that computes a maximally permissive program, but only considers *bad transitions* as candidates for removal.

Side effects. Implementability restrictions require that when we remove a transition t we also remove all other equivalent transitions $[t]_{LG}$. As a result, the removal of a bad transition might *introduce* additional bad transitions.

Definition 2. *We say that a removal of a transition t has a* side effect *when $|[t]_{LG}| > 1$. When the removal of a transition t does not have a side-effect, we say that it is* side-effect free.

Example 2. Consider the example program of Fig. 1 and its transition system in Fig. 2. Assume that the algorithm has chosen to remove the bad transition $\langle e, 2, 3, 1, 0, 0 \rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1 \rangle$, denote it t. The statement executed by this transition is $3\colon true \rightarrow$ z=y+1. Under observability limited to variables x, z, this removal has the *side effect*

of removing the (equivalent) transitions from $\langle e, e, 3, 1, 1, 0 \rangle$ and $\langle e, e, 3, 1, 2, 0 \rangle$. Since there are no other outgoing transitions from these states, the removal of t makes these states doomed, thus adding bad transitions.

Because the removal of a bad transition can introduce additional bad transitions (by introducing doomed and stuck states), an algorithm based on selecting bad transitions has to remove transitions gradually, and recompute the set of bad transitions after every step. This leads to the following algorithm.

```
EXHAUSTIVE(P : Program) : Program {
1:   R = ∅
2:   while (true) {
3:       ts = ⟨Σ_P, T_{P,S} \ R, Init_P⟩
4:       if valid(ts) return implement(P,R)
5:       B = bad-transitions(ts)
6:       if B = ∅ abort "cannot find valid synchronization"
7:       select a transition t ∈ B
8:       R = R ∪ [t]_{LG}
     }
}
bad-transitions(ts : TransSys) : Set of Transitions {
   let ts be ⟨Σ, T, Init⟩ in
   return {t ∈ T | i ⤳_{ts} src(t), dst(t) ⤳_{ts} d, i ∈ Init, d ∈ Doomed_{ts}}
}
```

Fig. 5. EXHAUSTIVE algorithm

Fig. 5 shows the EXHAUSTIVE algorithm for inferring synchronization. The algorithm takes a program as input and constructs a valid program by iteratively eliminating bad transitions. The algorithm maintains a set R of transitions to be removed. Initially, this set is empty. On every iteration of the algorithm, we construct a transition system ts by removing the transitions in R from the transition system of the input program P. If the resulting transition system is valid, the algorithm uses the procedure implement to return a modified version of P that avoids all transitions in R. If the transition system ts is not valid, the algorithm computes a set of bad transitions by using the procedure bad-transitions(ts). If the set is empty, it means that the transition system is not valid, but there are no more bad transitions to be removed (in this algorithm, it means that no bad transitions remain in ts and there exists a stuck state in *Init*). If the set B of bad transitions is not empty, the algorithm non-deterministically chooses one of the transitions in B as the transition to be removed. To guarantee that a program that avoids transitions in R is implementable, when we add a bad transition t to R, we add to R all transitions in its equivalence class $[t]_{LG}$.

Theorem 2 (Correctness of EXHAUSTIVE). *A run of the* EXHAUSTIVE *algorithm terminates with either a valid program or abort.*

Example 3. This example demonstrates how the algorithm is applied to the program of Fig. 1 and its transition system in Fig. 2. The first step in the algorithm is to check whether $ts = \langle \Sigma_P, T_{P,S} \setminus R, Init_P \rangle$ is valid. Since at this point $R = \emptyset$, the transition system is the one of Fig. 2 which is invalid (there is a trace reaching the stuck state $\langle e, e, e, 1, 2, 1 \rangle$). The algorithm now computes the set B, and lets assume that it chooses to remove the bad transition $t = \langle e, 2, 3, 1, 0, 0 \rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1 \rangle$. The statement executed by this transition is the statement 3: $true \rightarrow$ z=y+1. Under full observability, $\chi(src(t)) = (x == 1 \wedge y == 0 \wedge z == 0)$. Using this formula, the algorithm creates a new program P' in which the statement has the guard $\neg\chi(src(t))$, that is, 3: $(x \neq 1 \vee y \neq 0 \vee z \neq 0) \rightarrow$ z=y+1.

Next, we show how to use the EXHAUSTIVE algorithm to compute all maximally permissive programs for a given input program, specification and language of guards. The idea is to implement the non-deterministic choice of a transition $t \in B$ in line 7 using backtracking. As a result, we obtain different sets of transitions to remove, where each set yields a valid program. (It is different from enumerating all possible subsets of bad transitions of the original program, because of side effects.) The following lemma shows that all maximally permissive programs can be obtained this way.

Lemma 2. *For every maximally permissive program $P' \in MP(P, LG)$, there exists a run of the EXHAUSTIVE algorithm that returns P'' such that $[\![P']\!] = [\![P'']\!]$.*

Let PS denote the set of (valid) programs obtained from all possible runs of EXHAUSTIVE, for different choices of $t \in B$ in line 7. To compare permissiveness of programs $P_1, P_2 \in PS$, we look at the corresponding sets of removed transitions $R_1, R_2 \subseteq T_{P,S}$, computed by the EXHAUSTIVE algorithm, where $P_i = \texttt{implement}(P, R_i)$, for $i = 1, 2$. If $R_1 \subset R_2$, then the transition system obtained by removing R_1 has more traces (is more permissive) than the transition system obtained by removing R_2. Formally, let *RS* be the set of sets of removed transitions that correspond to the programs in PS. We define the operation $\min(RS)$ that chooses from RS the minimal sets of transitions that guarantee a valid transition system:

$$\min(RS) \stackrel{\text{def}}{=} \{R \in RS \mid \forall R' \in RS.R' \not\subset R\} \qquad (4)$$

This allows us to generate all maximally permissive programs:

Theorem 3. *For every maximally permissive program $P' \in MP(P, LG)$, there exists $R \in \min(RS)$ such that $[\![P']\!] = [\![\texttt{implement}(P, R)]\!]$. For every $R \in \min(RS)$, $\texttt{implement}(P, R) \in MP(P, LG)$.*

Complexity. A single run of EXHAUSTIVE is polynomial in the size of the (original) transition system. The size of RS is exponential in the transition system. Computing $min(RS)$ is polynomial in the size of RS. Therefore, computing $MP(P, LG)$ is exponential in the size of the transition system.

3.5 GREEDY Algorithm

The EXHAUSTIVE algorithm of Fig. 5 is choosing transitions for removal from the set bad-transitions(ts). This set may also contain transitions from one doomed state to another. Removal of a transition between doomed states is redundant, as such a transition

will become unreachable (and therefore transitively removed) when transitions into dominating doomed states are removed. We can further leverage the structure of the transition system and avoid removal of a transition between doomed states by having the algorithm pick transitions from the cut between non-doomed and doomed states.

The GREEDY algorithm is a modification of the EXHAUSTIVE algorithm such that instead of using bad-transitions(ts), it uses the following procedure cut-transitions(ts).

```
cut-transitions(ts : TransSys) : Transitions {
  let ts be ⟨Σ, T, Init⟩ in
  return {t ∈ T | i ⤳_ts src(t), i ∈ Init, src(t) ∉ Doomed_ts,  dst(t) ∈ Doomed_ts}
}
```

Example 4. Consider the program of Fig. 1. Assume *LG* is as earlier boolean combinations of equality to constants, and is limited to only observing variables x and z. The starting point of the algorithm is the transition system of Fig. 2. In the first step, the only transition in the cut is the transition $t = \langle e, 2, 3, 1, 0, 0\rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1\rangle$, and so the algorithm chooses to eliminate this transition. This results in the addition of the guard $(x \neq 1 \vee z \neq 0)$ to the statement z=y+1, and has the side-effect of removing the transitions from $\langle e, e, 3, 1, 1, 0\rangle$ and $\langle e, e, 3, 1, 2, 0\rangle$, which now become doomed states. In the second step, the algorithm chooses to eliminate the transition $\langle 1, e, 3, 0, 1, 0\rangle \xrightarrow{x=z+1} \langle e, e, 3, 1, 1, 0\rangle$. This adds the guard $(x \neq 0 \vee z \neq 0)$ to the statement x=z+1, which has the side effect of removing the transition $\langle 1, 2, 3, 0, 0, 0\rangle \xrightarrow{x=z+1} \langle e, 2, 3, 1, 0, 0\rangle$. The resulting program is shown in Fig. 3(b).

Theorem 4 (Correctness of GREEDY). *A run of the GREEDY algorithm terminates with either a valid program or abort.*

In GREEDY, the non-deterministic choice of a transition t at line 7 of Fig. 5 returns a cut transition. In contrast to EXHAUSTIVE, where the non-deterministic choice is implemented using backtracking, in GREEDY we implemented it as an arbitrary choice, because any choice returns a reasonable (in fact, locally optimal) solution, while enumerating all possibilities does not guarantee maximal permissiveness. Finding a maximally-permissive solution is exponential in the size of the transition system in the worst-case (using EXHAUSTIVE with backtracking), while GREEDY is polynomial. GREEDY computes a maximally permissive solution when the removal of transitions has no side-effects:

Theorem 5. *If a run of GREEDY has no side-effects then it computes a maximally permissive program for P and LG or aborts. If it aborts, then $MP(P, LG) = \emptyset$.*

Note that the theorem only requires that transitions removed during the run of GREEDY to be side-effect free. Recall that under full observability, there cannot be any side-effects, but GREEDY does not require full observability. That is, even under limited observability, it is possible that a run of GREEDY has no side-effects, in which case, it produces a maximally permissive program. However, in cases where limited observability causes side-effects, there are no guarantees: GREEDY may fail or succeed in finding a maximally permissive solution. The following example demonstrates that GREEDY fails to find a maximally permissive program when EXHAUSTIVE manages to find it.

Example 5. Consider the program of Fig. 1, and its transition system in Fig. 2. For this program, when the guard language is limited to only allow the observability of the variable z, the result of GREEDY is a program that admits no traces. However, the EXHAUSTIVE algorithm does find a solution with this guard language. The solution found by EXHAUSTIVE is the addition of a guard $z \neq 0$ to the statements x=z+1 and z=y+1.

In most examples we considered, even when GREEDY encountered side-effects, it was always able to find the best solution. Characterizing more accurately when GREEDY guarantees maximal permissiveness is an interesting subject of future work.

3.6   Challenges in Inferring Synchronization under Abstraction

The algorithms presented in this paper operate on a finite transition system. To handle infinite-state systems, we use finite-state abstraction. Given a program P and a specification S, we first compute an abstract transition system for it (see, e.g., [4]), and then apply EXHAUSTIVE or GREEDY to it. If the algorithm does not abort, then the resulting program is guaranteed to satisfy S. However, under abstraction, we cannot guarantee that the resulting program does not reach a stuck state. That is, we might generate guards that make a thread block indefinitely. The reason for this limitation is that under abstraction we might lose the information that a state becomes stuck.

We can conservatively eliminate abstract states that potentially become stuck, losing the ability to guarantee that the result is maximally-permissive. In many cases the conservative approach does not manage to find even a single valid program and aborts. Another approach is to refine an abstract transition system when a state becomes potentially stuck. In the case that the concrete transition system has a finite bisimulation quotient, our algorithm terminates and produces a valid program (or abort). Yet another approach is to use an abstraction that record information about stuck states. There are abstractions that can record some progress properties, but their precision for detecting stuck states has not been evaluated. This is a challenging problem, but it is beyond the scope of this paper.

4   Prototype Implementation

We have implemented the GREEDY algorithm in a prototype tool based on the SPIN model-checker [8]. The tool takes as input a program P, which uses CCRs, a specification S and a set of variables $Obs \subseteq Var$ that guards may refer to. The set of variables Obs is used to determine an upper bound on the synchronization cost. The tool then automatically infers correct synchronization with minimal cost, using the cost function from Section 4.1.

We used the tool on several small but instructive examples, described in [16], including dining philosophers and asynchronous counters. In all of the examples we start with a program that is initially incorrect and does not use any synchronization (our tool also works on input programs that already contain guards). In all examples, out tool successfully inferred guards that achieve maximal permissiveness.

4.1   Reducing Synchronization Cost

The algorithms presented so far infer correct (and maximally permissive) guards whose cost is less than a user-specified upper bound, however, the guards they produce are

not guaranteed to have the least synchronization cost for this level of permissiveness. Sometimes, it may be possible to reduce the cost of these guards while maintaining correctness and maximal permissiveness. We now demonstrate how this is done for a specific cost model.

Cost as the Number of Shared Accesses. Depending on the environment and the underlying architecture (e.g. cache costs), there may be different cost models for comparing the synchronization cost of two guard expressions. Here, we consider one intuitive cost model: we compare the number of distinct shared variables accessed in each guard. This is a natural measure reflecting the atomic observations about the shared state.

Formally, given a program P, we denote the number of distinct variables accessed by the CCR guard in location l of P by $nga(P, l)$. Given a program P, and a specification S, we say that $P_1 \in VP(P, S)$ has *lower cost* than a program $P_2 \in VP(P, S)$ if for every location l of P, $nga(P_1, l) \leq nga(P_2, l)$.

The language of guards is restricted to boolean combinations of equalities between a variable in the user-provided set *Obs* and an (integer) constant. We denote this language of guards by $EQ(Obs)$ and define a characterization function χ as follows:

$$\chi_{Obs}(s) \stackrel{\text{def}}{=} \bigwedge_{v \in Obs, [\![v]\!](s) = c} v = c$$

It is easy to see that χ_{Obs} is well defined and characterizes the states observable by the language defined above. The characterization function χ_{Obs} can be extended naturally to apply to sets of states. Given a set of states $S \subseteq \Sigma$, $\chi_{Obs}(S) = \bigvee_{s \in S} \chi_{Obs}(s)$.

The simple version of `implement` shown in Fig. 4 uses χ which finds a guard in the language, but does not attempt to minimize its cost. Synchronization derived using simple version of `implement` always has the same high cost: for each label l, $nga(P, l) = |Obs|$. Our tool uses an improved version of `implement`, shown in Fig. 6, which results in a program with the same permissiveness as for the simple version of `implement`, but has minimal cost. The main idea is to replace χ_{Obs} with a "separator" formula, as we briefly describe next (see [16] for details).

```
implement(P:Program,R:Transitions) {
  P' = P
  ts = ⟨Σ_P, T_{P,S} \ R, Init_P⟩
  L = {lbl(t) | t ∈ R}
  foreach l ∈ L
    BS={src(t)∈Reach_ts | lbl(t) = l, t∈R}
    GS={src(t)∈Reach_ts | lbl(t) = l}
    sep = SEPARATOR(BS, GS)
    let ccr(l) be guard → stmt in
    P' = P'[l : ¬sep ∧ guard → stmt]
  return P'
}
SEPARATOR(S_1, S_2 : Set of States) {
  foreach k = 1, . . . , |Obs|
    foreach V ⊆ Obs such that |V| = k
      if (S_1 ↓ V) ∩ (S_2 ↓ V) = ∅
        return χ_V(S_1);
  abort "cannot find separator"
}
```

Fig. 6. `implement` with separator

Separator. A separator is a guard in *LG* that distinguishes between two sets of states. Given $S_1, S_2 \subseteq \Sigma$, separator g satisfies (i) for all $s_1 \in S_1$, $[\![g]\!](s_1) = true$, and (ii) for all $s_2 \in S_2$, $[\![g]\!](s_2) = false$.

There may be multiple separators in *LG*, with different costs: for example $\chi_{Obs}(S_1)$ is a separator. However, the cost of this separator may be higher than necessary, because

it does not take into account S_2. The algorithm in Fig. 6 computes a separator in the language $EQ(Obs)$ with the minimal number of variables. It enumerates subsets V of Obs of increasing size until it finds one that can distinguish between S_1 and S_2, and builds a separator formula using χ_V.

The variables in V cannot distinguish between two states s and s' when the values of all these variables are identical in s and s'. Technically, we use $s \downarrow V$ to denote the projection of the state s onto the set of variables $V \subseteq Obs$. For a set $S \subseteq \Sigma$, we use $S \downarrow V$ to denote $\{s \downarrow V \mid s \in S\}$. A set of variables V can distinguish between sets of states S_1 and S_2, if their projections onto V are disjoint.

Example 6. Let $Var = \{x, y, z\}$. Let $S_1 = \{\langle 1, 1, 1\rangle\}$ and $S_2 = \{\langle 1, 1, 2\rangle, \langle 1, 2, 3\rangle\}$. Suppose that $Obs = \{x, z\}$. Then, a possible separator for S_1 and S_2 is $\chi_{Obs}(\langle 1, 1, 1\rangle)$ $\stackrel{\text{def}}{=} x = 1 \wedge z = 1$, which performs two shared accesses. Another separator for S_1 and S_2 is $z = 1$, and it only accesses a single variable. The algorithm in Fig. 6 returns the latter.

5   Related Work

Early work by Emerson and Clarke [2] uses temporal specifications to generate a synchronization skeleton. The generated programs assume full observability of the program state. This has been later extended by Attie and Emerson to synthesize programs with finer grained atomic sections [1]. Early work by Manna and Wolper [11] synthesizes CSP programs. In contrast, we synthesize programs for shared memory. These approaches have no notion of optimality, and no notion of synchronization cost. Our approach allows us to phrase the question of synchronization cost and optimality relative to a given language of guards. We also assume that the computation performed by the program is provided, and the goal of the synthesis algorithm is to add the required synchronization that guarantees that the specification is satisfied. Pnueli and Rosner [12] consider the problem of synthesizing a reactive module based on an LTL specification. They discuss the problem of *implementability* in this setting, and define necessary and sufficient conditions for the implementability of a given specification.

The work of Joshi et. al. [10] discusses a method for proving that a given program P is maximally concurrent (permissive) with respect to a specification S. This requires a manual phase where the input program P is translated to another equivalent program P' and maximal concurrency is then manually proved on P'. In contrast, we recognize that maximal concurrency is only one component of a more general problem that involves other important dimensions such as synchronization cost. We study how both of these two dimensions are connected and provide algorithms that take into account both dimensions when inferring synchronization.

References

1. Attie, P.C., Emerson, E.A.: Synthesis of concurrent systems for an atomic read/atomic write model of computation. In: PODC 1996, pp. 111–120. ACM Press, New York (1996)
2. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, pp. 52–71 (1982)

3. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
4. Dams, D.: Abstract Interpretation and Partition Refinement for Model Checking. PhD thesis, Eindhoven University of Technology, The Netherlands (December 1996)
5. Hansen, B.: Edison - a multiprocessor language. Software - Practice and Experience 11(4), 325–361 (1981)
6. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA 2003, pp. 388–402. ACM Press, New York (2003)
7. Hoare, C.A.R.: Towards a theory of parallel programming. In: The origin of concurrent programming: from semaphores to remote procedure calls, pp. 231–244 (2002)
8. Holzmann, G.J.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)
9. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
10. Joshi, R., Misra, J.: Toward a theory of maximally concurrent programs (shortened version). In: PODC 2000, pp. 319–328. ACM Press, New York (2000)
11. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. 6(1), 68–93 (1984)
12. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989, pp. 179–190. ACM Press, New York (1989)
13. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. 25(1), 206–230 (1987)
14. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: PPoPP 2007, pp. 271–271. ACM Press, New York (2007)
15. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue 3(7), 54–62 (2005)
16. Vechev, M., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. Technical report, IBM (2008), http://www.research.ibm.com/paraglide/

The Complexity of Predicting Atomicity Violations[*]

Azadeh Farzan[1] and P. Madhusudan[2]

[1] University of Toronto
[2] Univ. of Illinois at Urbana-Champaign

Abstract. We study the prediction of runs that violate atomicity from a single run, or from a regular or pushdown model of a concurrent program. When prediction ignores all synchronization, we show predicting from a single run (or from a regular model) is solvable in time $O(n+k.c^k)$ where n is the length of the run (or the size of the regular model), k is the number of threads, and c is a constant. This is a significant improvement from the simple $O(n^k \cdot 2^{k^2})$ algorithm that results from building a global automaton and monitoring it. We also show that, surprisingly, the problem is decidable for model-checking recursive concurrent programs without synchronizations. Our results use a novel notion of a *profile*: we extract profiles from each thread locally and compositionally combine their effects to predict atomicity violations.

For threads synchronizing using a set of locks $\mathcal{L}$, we show that prediction from runs and regular models can be done in time $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k + k^2})$. Notice that we are unable to remove the factor k from the exponent on n in this case. However, we show that a faster algorithm is *unlikely*: more precisely, we show that prediction for regular programs is unlikely to be fixed-parameter tractable in the parameters $(k, |\mathcal{L}|)$ by proving it is $W[1]$-hard. We also show, not surprisingly, that prediction of atomicity violations on recursive models communicating using locks is undecidable.

1   Introduction

The new disruptive trend in microprocessor technology that bodes a future where there will be no significant speed-up of individual processors but only a multitude of processor cores, poses a tremendous challenge to computer science. Parallel computers will become ubiquitous and all software will have to exploit parallelism to gain performance. One of the most challenging aspects of this overhauling of technology is that concurrent programs are very hard to write and debug, making reliability and programmer productivity a huge concern.

Despite various efforts in computer science that strive to enable simple models for concurrency such as *transactional memory*[24], stream-programming, actors and MPI (message passing interface) paradigms [1,14,2], that escape the dread of a wild shared-memory program, it is fairly clear that concurrent reactive

---

[*] For a full version of this paper refer to [8].

programs will exhibit significant non-determinism in terms of interleaved execu-
tions. A serious consequence of this is that software will become very hard even
to *test against one particular input*: given a concurrent program and an input,
there will be a myriad of interleaved executions, making testing extremely chal-
lenging. The CHESS project at Microsoft research and IBM's ConTest tool are
efforts that try to address this problem.

An extremely common *generic* concurrency bug is the violation of *atomicity*.
Intuitively, a programmer writing a procedure often wants non-interfered access
to certain data, enabling local reasoning of the procedure in terms of how it
affects the state. A programmer often puts together concurrency control mech-
anisms to ensure atomicity, often by taking locks on the data accessed. This is
extremely error-prone: errors occur if not all locks for accessed data are taken,
non-uniform ordering of locking can cause deadlocks, and naive ways of locking
can inhibit concurrency, which forces programmers to invent intricate ways to
achieve concurrency and correctness at the same time. Recent studies of concur-
rency errors [19] show that a majority of errors (69%) are atomicity violations.
This motivates the problems we consider in this paper: to study algorithms that
can help search the space of all interleavings for atomicity violations.

First, we assume that we have a mechanism to observe the global run of a con-
current program as an interleaved sequence of events executed by the different
threads[1]. Assuming a program's global run is divided into transactions, where a
transaction is a block of code like a procedure that we expect the programmer
intends to be atomic, we would like to check for runs of the program that violate
atomicity with respect to these transaction boundaries[2]. The notion of atom-
icity we study is a standard notion called *conflict-serializability*— intuitively, a
conflict serializable run is a run that may involve interleaving of threads but is
semantically equivalent to a serial run where all transactions are executed in a
sequential non-interleaved fashion.

Given a run, the first problem of interest is to check whether it is serializable.
This problem is a *monitoring* problem and we have recently solved this problem
satisfactorily [7], showing that there is a deterministic monitoring algorithm that
uses space at most $O(k^2 + kv)$ (for a program with k threads and v global vari-
ables). The salient aspect of this algorithm is that the space used is independent
of the length of the observed run, making it extremely useful in practice.

In this paper, we study the harder problem of *predicting* atomicity violations.
Given a run r, we would like to predict other runs r' which are not serializable.
This is an extremely interesting and useful problem to solve; if we execute a
program on an input and obtain one run r, and use it to predict non-serializable
runs r' *efficiently*, it gives us a very effective mechanism of finding atomicity vio-
lations without generating and testing all interleavings in a brute-force manner.

---

[1] In practice, we can augment the program so that it communicates to a monitoring
module, and with extra synchronization, ensure sequential consistency, and correct
observation of runs.

[2] Note that we do not assume a transactional memory programming model; the pro-
grams we consider run on "wild" shared memory.

Our prediction model is simple and intuitive: given a run r, we project the run r to each of the threads to get local runs $r_1, \ldots, r_k$. We then consider *all* runs that can be obtained by combining the runs r_1 through r_k in any interleaved fashion to be predicted by the run r. Note that our notion of a run does not include conditional checks made by the threads nor the actual data written by the programs: this is intentional, as considering these aspects leads to a very complex prediction model that is unlikely to be tractable. Our prediction model is *optimistic*: we predict a larger class of runs than may be allowed by the actual program, and hence any non-serializable execution that we infer must be subject to testing to check feasibility of execution by the program.

The problem of inferring whether any interleaved execution of k local runs $r_1, \ldots, r_k$ leads to a violation of serializability is really a *model-checking* problem: for each thread T_i we are given a *straight-line* program executing r_i, and asked whether the concurrent program has a serializability violation. A natural analog of this problem is that we are given a set of k program models (finite-state transition systems or recursive transition systems) and asked whether any interleaving of them results in a serializability violation. Program models can be derived in various ways: for instance we can collect the projections of *multiple* tests and build local transition systems and check whether we can predict a run that violates atomicity. Program models may also be obtained *statically* from programs using abstraction techniques.

This paper is devoted to the theoretical analysis of predicting atomicity violations from *straight-line concurrent programs* (for predictions from tests), *regular concurrent programs* and *recursive concurrent programs*.

Let us briefly consider the problem of inferring runs from straight-line. It is clear that we can construct a global transition system that generates all the interleavings of the program, and by intersecting this with a *monitoring automaton for serializability*, predict atomicity violations. However, this essentially generates all the interleavings, which is precisely the problem we wish to avoid. The goal of this paper is to study when this can be avoided.

Notice that the state space of the global transition system generating all interleavings is $O(n^k)$ in size where n is the size of the program, and k is the number of threads. In practical applications, n is very large (the length of the run) and k, though small, is not a constant, leading to a very large state-space, making prediction almost impossible. Moreover, we clearly cannot expect algorithms to work without an exponential dependence on k (we can show that the problem is NP-complete). However, it would be extremely beneficial if we can build algorithms *where k does not occur in the exponent on n*. An algorithm that works in time $O(n + k.c^k)$ would work much faster in practice. For instance, in the SOR benchmark (see [7]) for $k = 3$ threads, the length of a run is $n = 97 \times 10^6$ nodes, and nothing short of a linear dependency on n can really work in practice.

Secondly, predicting runs gets harder when the synchronization mechanisms have to be respected. In this paper, we consider two models: one where we ignore any synchronization mechanism (which leads to faster but less accurate predictions) and one where we consider synchronization using locks.

Our main contributions in this paper are the following. Assuming the set of variables manipulated is a constant, we show:

- For prediction without considering of synchronization mechanisms, we show:
  - Straight-line programs and regular programs over a fixed set of global variables are solvable in time $O(n + k.c^k)$ for a constant c (which depends quadratically on the number of variables). This result is proved by giving a *compositional* algorithm that extracts relevant results from each thread, using a novel notion called a *profile*, and combines the profiles to check violations.
  - Prediction of atomicity errors for recursive programs is (surprisingly) decidable, and can be done in time $O(n^3 + k.c^k)$.
- For prediction in programs that use lock synchronization over a lock-set $\mathcal{L}$:
  - Straight-line programs and regular programs over a fixed set of global variables are solvable in time $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k + k^2})$. This is a global algorithm that considers all interleavings, and hence k does occur in the exponent on n. However, we show that removing the k from the exponent is highly unlikely. More precisely, we show that it is unlikely that there is an an algorithm that works in time $O(\text{poly}(n) \cdot f(k, |\mathcal{L}|))$, for *any computable function* f, by showing that the problem is W[1]-hard over the parameters $(k, |\mathcal{L}|)$. W[1]-hard problems are studied in complexity theory, and are believed not to be fixed-parameter tractable.
  - Prediction of atomicity for recursive programs is (not surprisingly) undecidable.

Two aspects of our work are novel. First, the notion of profiles that we use to give the first sound and complete compositional mechanisms to prove atomicity of programs without locks. Second, for programs with locks, our W[1]-hardness lower bound shows that an efficient compositional method is unlikely. Such fixed-parameter intractability results are not common in the verification literature (we know of no such hardness result directly addressing model checking of systems).

The paper is organized as follows. In Section 2 we first define schedules which capture how programs access variables, then define the three classes of programs we study, namely straight-line, regular, and recursive programs. We also define the notion of conflict-serializability and its algorithmic equivalent in terms of conflict-graphs. Section 3 is devoted to the study of finding atomicity violations in programs with no synchronization mechanisms while Section 4 studies the problem for programs with lock synchronization. We end with concluding remarks and future directions in Section 5.

Related Work: Atomicity is a new notion of correctness for concurrent programs. It has been suggested [10,11,26,25,27] that *atomicity violations based on serializability* are effective in finding concurrency bugs. A recent and interesting study of bug databases identifies atomicity violations to be the single major cause for errors in a class of concurrent programs [19] Work in software verification for atomicity errors are often based on the Lipton-transactional framework. Lipton transactions are *sufficient* (but not necessary) thread-local conditions

that ensure serializability [18]. Flanagan and Qadeer developed a type system for atomicity [10] based on Lipton transactions (which, being local, is also compositional). Model checking has also been used to check atomicity using Lipton's transactions [11,15]. In [6], we had proposed a slightly different notion of atomicity called *causal atomicity* which can be checked using partial-order methods.

The run-time *monitoring* for atomicity violations is well-studied. Note that here the problem is to simply observe a run and check whether that particular run is atomic (involves no *prediction*). In a recent paper [9], the authors show monitoring algorithms that work with efficient space constraints to monitor atomicity violations during testing. In another recent paper [7], we have established a more sophisticated algorithm that uses *bounded* space to monitor, and results in extremely efficient monitoring algorithms. The existence of a monitor also implies that if the global state-space of a concurrent program can be modeled as a finite-state system, then the *model checking* problem for serializability is decidable.

The work in [22] defines *access interleaving invariants* that are certain patterns of access interactions on variables, learns the intended specifications using tests, and monitors runs to find errors. A variant of dynamic two-phase locking algorithm [20] for detection of an serializability violation is used in the atomicity monitoring tool developed in [27].

Turning to predictive analysis, there are two main streams of work that are relevant. In papers [26,25], Wang and Stoller study the prediction of runs that violate serializability from a single run. Under the assumptions of deadlock-freedom and nested locking, they show precise algorithms that can handle serializability violations involving *at most two transactions*. They also give heuristic incomplete algorithms for checking arbitrary runs. In contrast, the algorithms we present here do not make these assumptions, and are precise and complete. Predicting alternate executions from a single run are also studied in a series of papers by Rosu et al [23,4]. While these tools can also predict runs that can violate atomicity, their prediction model is tuned towards *explicitly* generating alternate runs, which can then be subject to atomicity analysis. In sharp contrast, the results we present here search the space of alternate interleavings efficiently, without enumerating them. However, the accuracy and feasibility of prediction in the above papers are better as the algorithm involves looking at the static structure of the programs and analyzing their control dependencies.

2 Modeling Runs of Concurrent Programs

A program consists of a set of threads that run concurrently. Each thread sequentially runs a series of *transactions*. A transaction is a sequence of actions; each action can be a read or write to a (global) variable.

We assume a finite set of thread identifiers $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$. We also assume a finite set of entity names (or just entities) $\mathcal{X} = \{x_1, x_2, \ldots, x_v\}$ that

the threads can access. Each thread $T \in \mathcal{T}$ can perform actions from the set $\mathcal{A}_T = \{T{:}\mathrm{read}(x), T{:}\mathrm{write}(x) \mid x \in \mathcal{X}\}$. Define $\mathcal{A} = \bigcup_{T \in \mathcal{T}} \mathcal{A}_T$.

For most parts of this paper (save for the technical lemmas), we will assume that the number of variables $|\mathcal{X}| = v$ is a fixed constant.

Let us define for each thread $T \in \mathcal{T}$, the extended alphabet $\Sigma_T = \mathcal{A}_T \cup \{T{:}\triangleright, T{:}\triangleleft\}$. The events $T{:}\mathrm{read}(x)$ and $T{:}\mathrm{write}(x)$ correspond to thread T reading and writing to entity x, $T{:}\triangleright$ and $T{:}\triangleleft$ correspond to boundaries that begin and end transactional blocks of code in thread T. Let $\Sigma = \bigcup_{T \in \mathcal{T}} \Sigma_T$.

For any alphabet A, $w \in A^*$, let $w[i]$ (where $i \in [0, |w| - 1]$) denote the i'th element of w, and $w[i, j]$ denote the substring from position i to position j (both inclusive) in w. For $w \in A^*$ and $B \subseteq A$, let $w|_B$ denote the word w projected to the letters in B. For a word $w \subseteq \Sigma^*$, $w|_T$ be a shorthand notation for $w|_{\Sigma_T}$, which denotes the actions that thread T partakes in.

The following defines the notion of observable behaviors on the global variables of a concurrent program, which we call a *schedule*.

Definition 1. *A transaction tr of a thread T is a word in $(T{:}\triangleright) \cdot \mathcal{A}_T^* \cdot (T{:}\triangleleft)$. Let $Tran_T$ denote the set of all transactions of thread T, and let $Tran$ denote the set of all transactions. A schedule is a word $\sigma \in \Sigma^*$ such that for each $T \in \mathcal{T}$, $\sigma|_T$ is a prefix of $Tran_T^*$. Let Sched denote the set of all schedules.*

In other words, the actions of thread T are divided into a sequence of transactions, where each transaction begins with $T{:}\triangleright$, is followed by a set of reads and writes, and ends with $T{:}\triangleleft$. Let *Sched* denote the set of all schedules.

When we refer to two particular events $\sigma[i]$ and $\sigma[j]$ in σ, we say they *belong* to the same transaction if they belong to the same transaction block: i.e. if there is some T such that $\sigma[i], \sigma[j] \in \mathcal{A}_T$, and there is no i', $i < i' < j$ such that $\sigma[i'] = T{:}\triangleleft$. We will refer to the transaction blocks freely and associate (arbitrary) names to them, using notations such as tr, tr_1, tr', etc.

Concurrent Programs

We now define the three classes of programs we will work with— straight-line, regular, and recursive programs.

For a set of locks $\mathcal{L}$, and thread $T \in \mathcal{T}$, define the set of lock-actions of T as $\Pi_{\mathcal{L},T} = \{T{:}acquire(l), T{:}release(l) \mid l \in \mathcal{L}\}$. Let $\Pi_{\mathcal{L}} = \bigcup_{T \in \mathcal{T}} \Pi_{\mathcal{L},T}$.

A word $\gamma \in \Pi_{\mathcal{L}}^*$ is *lock-valid* if it respects the usual locking pattern imposed by a locking mechanism, or formally, if for every $l \in \mathcal{L}$, $\gamma|_{\Pi_{\{l\}}}$ is a prefix of $\left[\bigcup_{T \in \mathcal{T}} (T{:}acquire(l) \ T{:}release(l))\right]^*$.

We consider three frameworks based on the structure of code in the threads.

- **A Straight-line program over** $\mathcal{L}$ is a set $Pr = \{\alpha_T\}_{T \in \mathcal{T}}$ where $\alpha_T \in (T{:}\triangleright(\mathcal{A}_T \cup \Pi_{\mathcal{L},T})^* T{:}\triangleleft)^*$ such that $\alpha_T|_{\Pi_{\mathcal{L},T}}$ is lock-valid.

  The runs defined by the program Pr is given by: $Runs(Pr) = \{w \mid w \in (\Sigma \cup \Pi_{\mathcal{L}})^*, \text{s.t. } w|_{\Pi_{\mathcal{L}}} \text{ is lock-valid and } w|_{\Sigma_T} \text{ is a prefix of } \alpha_T, \text{ for each } T \in \mathcal{T}\}$.

- **A regular program over** $\mathcal{L}$ is a set $Pr = \{A_T\}_{T \in \mathcal{T}}$ where each A_T is a finite transition system. $A_T = (Q_T, q_{in}^T, \rightarrow_T)$ where Q_T is a finite set of states, $q_{in}^T \in Q_T$ is the initial state, and $\rightarrow_T \subseteq Q_T \times (\Sigma_T \cup \Pi_{\mathcal{L},T}) \times Q_T$ is the transition relation. The language of A_T, $L(A_T)$, is the set of all words $w \in (\mathcal{A}_T \cup \Pi_{\mathcal{L},T})^*$ on which there is a path from q_{in} on w. We require that for any $w \in L(A_T)$, $w|_{\Pi_{\mathcal{L},T}}$ is lock-valid, and $w|_{\Sigma_T}$ is a prefix of $Tran_T^*$.

  The runs defined by Pr is given by:

  $Runs(Pr) = \{w|\ w \in (\Sigma \cup \Pi_{\mathcal{L}})^*, \text{s.t. } w|_{\Pi_{\mathcal{L}}}$ is lock-valid and for each $T \in \mathcal{T}, w|_{\Sigma_T} \in L(A_T)\}$.

- **A Recursive program over** $\mathcal{L}$ is a set $Pr = \{P_T\}_{T \in \mathcal{T}}$ where each P_T is a pushdown transition system $P_T = (Q_T, q_{in}^T, \Gamma^T, \rightarrow_T)$ where Q_T is a finite set of states, $q_{in}^T \in Q_T$ is the initial state, Γ^T is the stack alphabet, and $\rightarrow_T \subseteq Q_T \times (\Sigma_T \cup \Pi_{\mathcal{L},T}) \times \{\text{push}(d), \text{pop}(d), \text{skip}\}_{d \in \Gamma^T} \times Q_T$ is the transition relation. The language of P_T, $L(P_T)$ is the set of all words generated by P_T and is defined as usual. We again require that for any $w \in L(P_T)$, $w|_{\Pi_{\mathcal{L},T}}$ is lock-valid, and $w|_{\Sigma_T}$ is a prefix of $Tran_T^*$.

  The runs defined by Pr is given by: $Runs(Pr) = \{w|\ w \in (\Sigma \cup \Pi_{\mathcal{L}})^*,$ s.t. $w|_{\Pi_{\mathcal{L}}}$ is lock-valid and for each $T \in \mathcal{T}, w|_{\Sigma_T} \in L(A_T)\}$.

Finally, for any program Pr as above, the set of schedules defined by Pr is defined as $Sched(Pr) = Runs(Pr)|_{\Sigma}$. A program without locks is a program Pr over the empty set of locks.

Defining atomicity

We now define atomicity as the notion of *conflict serializability*. Define the *dependency* relation D as a symmetric relation defined over the events in Σ, which captures the dependency between (a) two events accessing the same entity, where one of them is a write, and (b) any two events of the same thread, i.e.,

$$D = \{(T_1{:}a_1, T_2{:}a_2) \mid T_1 = T_2 \text{ and } a_1, a_2 \in A \cup \{\triangleright, \triangleleft\} \text{ or}$$
$$\exists x \in \mathcal{X} \text{ such that } (a_1 = \text{read}(x) \text{ and } a_2 = \text{write}(x)) \text{ or}$$
$$(a_1 = \text{write}(x) \text{ and } a_2 = \text{read}(x)) \text{ or } (a_1 = \text{write}(x) \text{ and } a_2 = \text{write}(x))\}.$$

Definition 2 (Equivalence of schedules). *The equivalence of schedules is defined as the* smallest *equivalence relation* $\sim \subseteq Sched \times Sched$ *such that: if* $\sigma = \rho ab\rho', \sigma' = \rho ba\rho' \in Sched$ *with* $(a, b) \notin D$, *then* $\sigma \sim \sigma'$.

It is easy to see that the above notion is well-defined. Two schedules are considered equivalent if we can derive one schedule from the other by iteratively swapping consecutive independent actions in the schedule.

We call a schedule σ *serial* if all the transactions in it occur sequentially: formally, for every i, if $\sigma[i] = T{:}a$ where $T \in \mathcal{T}$ and $a \in A$, then there is some $j < i$ such that $T[i] = T{:}\triangleright$ and every $j < j' < i$ is such that $\sigma[j'] \in A_T$. In other words, the schedule is made up of a sequence of complete transactions from different threads, interleaved at boundaries only.
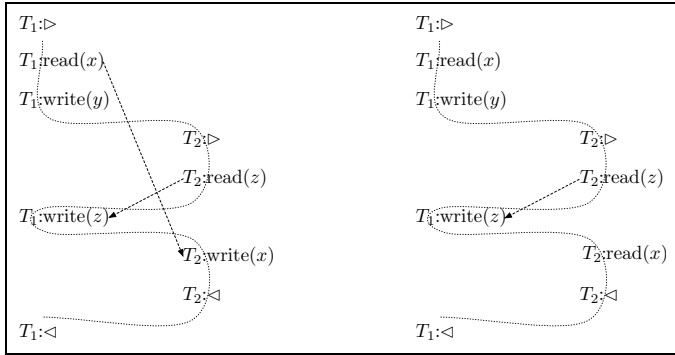
Fig. 1. A non-serializable schedule and a serializable schedule

Definition 3. *A schedule is* serializable *if it has an equivalent serial schedule. That is, σ is a serializable schedule if there a* serial *schedule σ' such that $\sigma \sim \sigma'$.*

Example 1. Figure 1 contains two schedules depicted by the dotted lines. The one on the left is not serializable. The dependent events $(T_1{:}\mathrm{read}(x), T_2{:}\mathrm{write}(x))$ indicate that T_2 has to be executed after T_1 in a serial run, while the pair of dependent events $(T_2{:}\mathrm{read}(z), T_1{:}\mathrm{write}(z))$ impose the opposite order. Therefore, no equivalent serial run can exist. The schedule on the right is serializable since in an equivalent serial run exists that runs T_2 followed by T_1.

The Conflict-Graph Characterization: For any schedule σ, let us give names to transactions in σ, say $tr_1, \ldots, tr_n$. The *conflict-graph* of σ is $CG(\sigma) = (V, E)$ where $V = \{tr_1, \ldots, tr_n\}$ and E contains an edge from tr to tr' iff there is some event a in transaction tr and some action a' in transaction tr' such that (1) the a-event occurs before a' in σ, and (2) aDa'.

Lemma 1. *[3,20,12,7] A schedule σ is atomic iff the conflict graph associated with σ is acyclic.*

The above characterization yields a simple algorithm for serializability:

Proposition 1. *The problem of checking whether a single schedule σ is serializable is decidable in polynomial time.*

3   Model Checking Atomicity for Concurrent Programs without Synchronizations

In this section, we present model checking algorithms for checking atomicity of finite-state concurrent programs (straight-line, regular, and recursive programs). Let us first show that if the program has a non-serializable run, then it has a non-serializable run of a particular form.

A run σ is said to be *normal* if there is a thread T_i such that $\sigma = u_i \cdot T_i{:}\triangleright \ \cdot v_i \cdot w_1 \cdot w_2 \cdots w_{i-1} \cdot w_{i+1} \cdots w_k \cdot v_i' \cdot T_i{:}\triangleleft \cdot u_i'$, where $w_j = \sigma|_{\Sigma_{T_j}}$ (for every j), $u_i \cdot T_i{:}\triangleright \cdot v_i \cdot v_i' \cdot T_i{:}\triangleleft \cdot u_i' = \sigma|_{\Sigma_{T_i}}$, and $v_i \cdot v_i' \in A_{T_i}^*$. In other words, a run is normal if it executes a thread from the beginning up to the middle of a transaction in that thread, executes other threads serially and completely, and then finishes the incomplete thread. The Figure on the right demonstrates the normal run which is equivalent to the non-serializable run in Figure 1 (on the left).
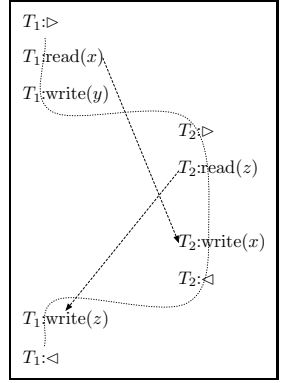
The following observation will prove useful throughout this section:

$T_1{:}\triangleright$
$T_1{:}\text{read}(x)$
$T_1{:}\text{write}(y)$
$T_2{:}\triangleright$
$T_2{:}\text{read}(z)$
$T_2{:}\text{write}(x)$
$T_2{:}\triangleleft$
$T_1{:}\text{write}(z)$
$T_1{:}\triangleleft$

Lemma 2. *If a program with no locks ($\mathcal{L} = \emptyset$) has a non-serializable run, then it has a non-serializable normal run.*

The crucial observation (behind Lemma 2) is that there are really at most two events in each thread that contribute to evidencing the cycle in the conflict graph, and hence witnessing non-serializability. Intuitively, for each thread T in the cycle, we pick can pick two events in_T and out_T, that cause respectively the incoming edge from the previous thread and the outgoing edge to the next thread in the cycle. This observation leads us to the following notion of profiles:

Definition 4 (Profile). *Let $\sigma_T \subseteq \Sigma_T^*$ be a local schedule. A profile for σ_T is a (bounded-length) word π that is of one of the following forms:*

- *$\pi = T{:}\triangleright \ T{:}a \ T{:}\triangleleft$, where $T{:}a$ occurs in σ_T, or*
- *$\pi = T{:}\triangleright \ T{:}a \ T{:}b \ T{:}\triangleleft$, provided there are two indices i and j such that $i < j$, $\sigma_T[i] = T{:}a$, $\sigma_T[j] = T{:}b$, and moreover there is no i' with $i < i' < j$ and $\sigma_T[i'] = T{:}\triangleleft$. In other words, $T{:}a$ and $T{:}b$ occur as events in σ_t in that order, and belong to the same transaction.*
- *$\pi = T{:}\triangleright \ T{:}a \ T{:}\triangleleft \ T{:}\triangleright \ T{:}b \ T{:}\triangleleft$, provided there are two indices i and j such that $i < j$, $\sigma_T[i] = T{:}a$, $\sigma_T[j] = T{:}b$, and moreover there is an i' with $i < i' < j$ and $\sigma_T[i'] = T{:}\triangleleft$. In other words, $T{:}a$ and $T{:}b$ occur as events in σ_t in that order, and belong to different transactions.*

The idea of a profile is that it picks one or two events from a thread's execution, along with the information as to whether the two events occurred in the same transaction or in different transactions. It turns out that profiles are enough to witness non-serializability.

Lemma 3. *A program P with no locks (straight-line, regular, or recursive) has a non-serializable run if and only if there exists a set $\langle \pi_T \rangle_{T \in \mathcal{T}}$, where each π_T is a profile of $\sigma|_T$, such that the straight line program defined by these profiles has a non-serializable run.*

The above lemma is very important, as it says that no matter how long or complex a thread is, we can summarize it using short profiles and check the profiles for non-serializability. This will form the key technical idea in proving the upper bounds in this section.

3.1   Straight-Line and Regular Programs

We discuss now the problem of checking whether a straight-line or regular program has a non-serializable schedule. We show that, by using profiles, we can solve this problem in $O(n + k.c^k)$ time where n is the maximum size of the program for any thread, k is the number of threads, and c is a constant.

Suppose that a regular program Pr consists of threads $T_1, \ldots, T_k$. The idea is to replace each thread T_i by a set of profiles $\mathbf{P}_i$, and then check whether the collection of profiles $\mathbf{P}_1, \ldots, \mathbf{P}_k$ induces a non-serializable run. By Lemma 3, Pr has a non-serializable run if and only if the collection of profiles $\mathbf{P}_1, \ldots, \mathbf{P}_k$ induces a non-serializable run.

For all threads T_i, the set of profiles $\mathbf{P}_i$ can be computed from T_i in $O(n)$ time. Assuming that T_i is represented by a finite transition system of size n, one can establish in time linear in n whether a profile π is a profile of T_i. Since there are at most v^2 possible profiles (where v is the number of global variables), one can compute all profiles of T_i in time $O(v^2 n)$. In fact, we construct an automaton P_i which accepts $\mathbf{P}_i$, the set of all profiles of T_i. These profile automata are all of size $O(v^2)$. We build a product automaton $\mathcal{P}$ which accepts the set of all possible interleavings of strings accepted by $P_1, \ldots, P_k$. Hence, $\mathcal{P}$ accepts the set of all possible interleavings of profiles of threads $T_1, \ldots, T_k$, and its size is $O\left((v^2)^k\right)$.

We can now intersect $\mathcal{P}$ with a monitoring automaton $\mathcal{S}$ for non-serializability (see [7]). The monitor maintains a graph with k nodes, and a set of labels of size $O(v)$ for each node; therefore $\mathcal{S}$ is of size $O\left(2^{k^2} + k2^{vk}\right)$. However, since it suffices to monitor only *normal* runs, we can restrict ourselves to graphs with only a linear number of edges and vertices, resulting in an automaton of size $O\left(k2^{vk}\right)$. Thus, the product automaton is of size $O\left(k.2^{vk}.v^{2k}\right)$. Hence we have the following result:

Theorem 1. *Given a straight-line or regular program Pr, one can check in time $O(nv^2 + k2^{vk}.v^{2k})$ whether Pr has a non-serializable run, where n is the maximum size of a thread, k is the number of threads, and v is the number of variables. When v is a constant, the complexity reduces to $O(n + kc^k)$ where c is a constant.*

We can show that, in general, an exponential dependence on the input is unlikely to be avoidable:

Theorem 2. *The problem of checking non-serializability of straight-line programs and regular programs, without locks, are both NP-complete.*

3.2   Recursive Programs

In this section, we discuss the effect of the presence of recursion in the code on the serializability checking problem. Note that even reachability of a global state is *undecidable* for concurrent recursive programs, and, since serializability is a fairly complex global property, even the decidability of serializability is not obvious.

We show, surprisingly, that checking serializability for recursive programs without locks is indeed *decidable* and in time $O(n^3 + k.c^k)$. Again, the notion of profiles come to the rescue, as they avoid searching the global state-space.

By Lemma 4, the witness for non-serializability need only contain a profile of each thread; Therefore, we can, similar to the regular program case, extract the profiles of each thread, and combine the profiles (which are straight-line programs) to check for non-serializability.

Extracting profiles from non-recursive threads is a rather straightforward task. For recursive programs, this is slightly more involved. Recall that each thread T is modeled as a pushdown automaton (PDA) P_T. We show that for any PDA P, we can *efficiently* construct an NFA (nondeterministic finite automaton) N, such that the set of profiles of P and N are the same. Therefore, we can replace the PDA model (the recursive code) of a thread by regular program, effectively removing recursion, and reduce serializability of recursive programs to that of regular programs.

Lemma 4. *For a PDA P, we can construct, in $O(|P|^3.v^2)$-time, an NFA that is of size $O(|P|.v^2)$ and that accepts the set of all profiles of schedules of P.*

The result below follows from our result on checking serializability of regular programs.

Theorem 3. *Given a recursive program Pr, the problem of checking whether it generates a non-serializable schedule, is solvable in time $O(n^3v^2 + k.2^{vk}.v^{2k})$, where n is size of the program, k is the number of threads, and v is the number of variables. When v is a constant, the complexity reduces to $O(n^3 + kc^k)$, where c is a constant.*

4   Programs with Lock Synchronization

In this section, we consider programs that synchronize using locks. We establish two simple results: first, we show that the problem of checking straight-line and regular programs with locks is solvable in time $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k})$, and, second, that the problem of checking recursive programs with locks is undecidable. Note that the complexity bounds we prove for straight-line programs and regular programs are *not* of the form $O(\text{poly}(n) \cdot 2^{|\mathcal{L}| \cdot \log k} \cdot f(k))$, i.e., we do not remove k from the exponent on n, as we did for checking atomicity of programs without locks by extracting profiles locally and combining them. However, for programs with locks, a notion of summarizing a thread using a finite amount of information that is independent of n seems hard. In fact, we believe that *no such scheme* exists. More precisely, we show that the problem of checking atomicity in regular programs with locks is unlikely to be *fixed-parameter tractable* (i.e., it is unlikely that there is an algorithm that works in time $O(\text{poly}(n) \cdot f(k, |\mathcal{L}|))$ for *any computable function f*) by showing that the problem is W[1]-hard.

Given a straight-line or regular program with locks, we can construct the product machine that generates all global runs. This machine will be of size $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k})$, as its state-space will track individual states of each thread, and in addition will keep track for each lock, the thread that holds it. We can now intersect this with a monitoring automaton for non-serializability (see [7]),

which is of size $O(2^{k^2+kv})$. It is easy to see that the language of the resulting automaton is empty if and only if the program has a serializability violation. We therefore have proven the following theorem.

Theorem 4. *The problem of checking whether a straight-line program or a regular program with locks has a serializability violation is decidable in time $O(n^k \cdot 2^{|\mathcal{L}|\cdot \log k+k^2+kv})$. When v is a constant, the complexity reduces to $O(n^k \cdot 2^{|\mathcal{L}|\cdot \log k+k^2})$.*

Let us now consider recursive programs with locks. It is known that the global reachability problem for two recursive machines communicating via synchronous messages is *undecidable* [21]. Moreover, it is known (see Kahlon et al [16]) that synchronous messages can be simulated using locks, and hence the global reachability problem for two recursive machines synchronizing using locks is undecidable. It is not hard to reduce this problem to checking serializability of a recursive program: intuitively, we augment the machines to execute a non-serializable run when they reach their respective goal states. Hence:

Theorem 5. *The problem of checking whether a recursive program with locks has serializability violations is undecidable.*

4.1   A Lower Bound on Checking Atomicity of Lock Synchronized Regular Programs

In this section, we will assume that the number of variables, v, is a constant.

In the setting of programs where all synchronization was ignored, we showed that predicting atomicity errors can be done in time $O(\text{poly}(n) \cdot k.c^k)$. As we argued, this is a much better algorithm than the naive algorithms that work in time $O(n^k)$ as typically n is much larger than k. In the setting of programs that synchronize using locks, we showed only an algorithm that runs in time $O(n^k \cdot 2^{|\mathcal{L}|\cdot \log k})$. A natural question is to ask whether this problem can also be solved in time $O(\text{poly}(n) \cdot 2^{|\mathcal{L}|\cdot \log k} \cdot f(k))$. We now show that this is unlikely: in fact, we show that the problem is unlikely to be *fixed-parameter tractable* (over the parameter k) by showing it is $W[1]$-hard.

Consider a problem X in which to each instance i we associate in addition to its size n a second a *parameter* $k \in \mathbb{N}$. Then the problem X is said to be *fixed-parameter tractable* with respect to k if there is an algorithm that decides X in time $O(n^c \cdot f(k))$, where f is an *arbitrary* function (we will assume f is computable) and c is a constant.

Fixed-parameter tractability is a mature area of computational complexity theory; we refer the reader to the textbooks [5,13]. For instance, finding a vertex cover of a graph G with k sets is an NP-complete problem, but is fixed-parameter tractable when the parameter is k (in fact, solvable in time $O(2^k \cdot |G|)$). Also, there is a hierarchy of classes of problems, called the W-hierarchy, for which no fixed-parameter tractable algorithms are known, and it is believed that problems complete for these classes are not fixed-parameter tractable. For instance, finding an independent set of size k in a graph G, where k is the parameter, is known to be $W[1]$-hard and hence not believed to be fixed-parameter tractable.

In this section, we will show that the problem of checking whether a regular program with locks has an atomicity violation, where the parameters are the number of threads in the program and the number of locks, is W[1]-hard.

We show hardness by reducing the problem of finite state automata intersection given below, which is known to be W[1]-hard, to our problem:

Finite State Automata Intersection
Instance: A set of k deterministic finite-state automata $A_1, \ldots A_k$ over a common alphabet Σ (Σ is *not* fixed).
Parameters: k, m
Question: Is there a string $w \in L(A_1) \cap L(A_2) \cap \ldots L(A_k)$ with $|w| \geq m$?

Given an instance of this problem $\langle A_1, \ldots, A_k \rangle$, we construct finite-state automata $B_1, \ldots B_k$ over a set of locks $\mathcal{L}$ and variables V such that they have a serializability violation if and only if the intersection of $A_1, \ldots, A_k$ is nonempty. Furthermore, and most importantly, $|\mathcal{L}| = O(k \cdot |\Sigma|)$, $V = \{x\}$, a single variable, and each B_i will be of size $O(|A_i| \cdot m)$. Note that the parameters never occur in the exponent in the complexity of any of these sizes. Hence, an FPT algorithm for serializability of regular programs with locks will imply that the finite-state intersection problem is fixed-parameter tractable, which is unlikely as it is W[1]-hard.

The construction proceeds in two phases. First, we construct automata $C_1, \ldots, C_k$ that communicate using pairwise rendezvous, and show that they exhibit a serializability violation if and only if the intersection of $A_1, \ldots, A_k$ is nonempty. Then we show that the pairwise rendezvous mechanism can be simulated using locks. Intuitively, the automaton C_1 guesses a letter and communicates it to all other processes by relay messaging. All automata update their state, each C_i simulating automaton A_i. C_1 ensures that at least m letters have been guesses, and then sends a message asking whether all other processes have reached their final states. If they all respond that they have, C_1 and C_2 perform a sequence of accesses to a single variable x that results in a serializability violation. Finally, we show that we can simulate the pairwise rendezvous of communication using only lock-synchronization (using a mechanism in Kahlon et al [16], and build automata $B_1, \ldots, B_k$ such that they exhibit a serializability violation if and only if the intersection of the languages of $A_1, \ldots, A_k$ has a string longer than m. This leads us to the following theorem:

Theorem 6. *The following problem:*

Serializability of Regular Programs
Instance: *A regular program $B_1, \ldots B_k$ with lock synchronization over a set of locks L and over a single global variable x.*
Parameter: $k, |L|$
Question: *Is the program atomic?*

is W[1]-hard.   □

The above shows that it is unlikely that there is an algorithm that can solve atomicity of regular programs in time $O(\text{poly}(n) \cdot f(k, |\mathcal{L}|))$. The question as to whether the problem of checking serializability violations of *straight-line* programs is also W[1]-hard is open.

The above reduction from automata intersection to atomicity has the property that the state-space of the machines and the lock-set are only linear in k; this has further implications. In [17], it was shown that the intersection of k finite-state automata, each of size n, is unlikely to be solvable in time $O\left(n^{(k/f(k))+d}\right)$ where $f = o(k)$ and $d > 0$ is a constant (i.e. reducing the exponent from k to a function sublinear in k). The authors show that if this were true, then problems solvable in nondeterministic time t would be solvable in subexponential deterministic time. This unlikelihood combined with our reduction (simplified not to count the number of letters in the word) implies that it is unlikely to find algorithms for atomicity that work in time $O\left(n^{(k/f(k))+d}\right)$ as well. That is, not only is k unavoidable in the exponent on n, a sub-linear exponent is also unlikely.

5 Conclusion and Future Work

We have established fundamental algorithms for predicting atomicity violations from straight-line programs, regular programs, and recursive programs. We have studied two prediction models: one which ignores any synchronization of the threads, and the other that considers lock-based synchronization. Our main results are that the problem is tractable, and solvable without exploring all interleavings, for the case when synchronizations are ignored. We believe that the notion of profiles set forth in this paper, which compositionally solve the serializability model-checking problem, will be very useful in practical tools. For synchronization using locks, we showed that such an efficient compositional scheme is unlikely, by proving a W[1]-hardness lower bound for regular programs.

There are several future directions worthy of pursuit. First, we are implementing prediction tools for atomicity violations in large programs, and preliminary results show that more restrictions (such as limiting violations to involve only two threads) are needed to make algorithms practical. Second, we do not know whether prediction of atomicity violations of straight-line programs with locks is also W[1]-hard; establishing this will give a strong argument to use prediction models that ignore synchronizations. Finally, the recent study of *nested locking* holds promise, as global reachability of concurrent programs synchronizing via nested locks admits a compositional algorithm [16]. We would like to investigate whether atomicity prediction can also benefit if threads use nested locking.

References

1. MPI: A message-passing interface standard,
   http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
2. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
3. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. 13(2), 185–221 (1981)

4. Chen, F., Serbanuta, T.F., Rosu, G.: jpredictor: a predictive runtime analysis tool for java. In: ICSE, pp. 221–230 (2008)

5. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1998)

6. Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)

7. 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)

8. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. Technical Report CSRG-591, University of Torotno, Department of Computer Science (2009)

9. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI, pp. 293–303 (2008)

10. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349 (2003)

11. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL, pp. 256–267 (2004)

12. Fle, M.P., Roucairol, G.: On serializability of iterated transactions. In: PODC 1982: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing, pp. 194–200. ACM Press, New York (1982)

13. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)

14. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS, pp. 151–162 (2006)

15. Hatcliff, J., Robby, Dwyer, M.B.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 175–190. Springer, Heidelberg (2004)

16. 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)

17. Karakostas, G., Lipton, R.J., Viglas, A.: On the complexity of intersecting finite state automata and n l versus n p. Theor. Comput. Sci. 302(1-3), 257–274 (2003)

18. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)

19. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, pp. 329–339 (2008)

20. Papadimitriou, C.: The theory of database concurrency control. Computer Science Press, Inc, New York (1986)

21. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)

22. Lu, S., Tucek, J., Qin, F., Zhou, Y.: Avio: detecting atomicity violations via access interleaving invariants. In: ASPLOS, pp. 37–48 (2006)

23. Sen, K., Rosu, G., Agha, G.: Online efficient predictive safety analysis of multithreaded programs. STTT 8(3), 248–260 (2006)

24. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)

25. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP, pp. 137–146 (2006)

26. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multi-threaded programs. IEEE Transactions on Software Engineering 32, 93–110 (2006)

27. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. 40(6), 1–14 (2005)

MoonWalker: Verification of .NET Programs

Niels H.M. Aan de Brugh[1], Viet Yen Nguyen[2], and Theo C. Ruys[3]

[1] OCÉ, Venlo, The Netherlands
niels.aandebrugh@oce.com
[2] RWTH Aachen University, Germany
http://moves.rwth-aachen.de/~nguyen/
[3] University of Twente, The Netherlands
http://www.cs.utwente.nl/~ruys/

Abstract. MoonWalker is a software model checker for CIL bytecode programs, which is able to detect deadlocks and assertion violations in CIL assemblies, better known as Microsoft .NET programs. The design of MoonWalker is inspired by the Java PathFinder (JPF), a model checker for Java programs. The performance of MoonWalker is on par with JPF. This paper presents the new version of MoonWalker and discusses its most important features.

1 Introduction

This paper presents MoonWalker[1] 1.0 [18], a software model checker for the verification of CIL bytecode programs. CIL stands for Common Intermediate Language and is the platform independent bytecode used within Microsoft's .NET. MoonWalker targets programs compiled against the MONO development platform [16], an open source implementation of the .NET development platform.

MoonWalker is a software model checker that uses the *virtual machine* approach of verification: the effect of every CIL bytecode instruction is analysed by the tool. MoonWalker systematically explores all reachable states of the application under verification, which involves executing bytecode instructions, storing and restoring states, and checking for safety properties. During exploration, MoonWalker will check for deadlocks and assertion violations.

The approach of MoonWalker is inspired by the Java PathFinder (JPF) [12,15], a very successful software model checker for Java. JPF pioneered the concept of implementing a software model checker around a virtual machine. And although the object-oriented design and the actual implementation of MoonWalker (in C#) and organisation of the classes and algorithms are different, all credits for the verification approach should go to the developers of JPF.

With MoonWalker 1.0, however, there is now a competitive software model checker readily available for the .NET framework. An important advantage of CIL over Java bytecode is that CIL has been designed to be the target for many programming languages, not just C#. See [17] for a complete overview. Finally, version 1.0 of MoonWalker incorporates some new techniques not yet available in other model checkers.

---

[1] MoonWalker was previously known as MMC: the Mono Model Checker.

XRT [5] is an alternative software model checker for .NET, which follows the same approach as JPF. XRT is not publicly available.

2   MoonWalker 1.0

The architecture of MoonWalker 0.5 has been outlined in [11] and the design and implementation of version 0.5 are described in detail in [1]. This section discusses the new features that have been added to MoonWalker over the past $1\frac{1}{2}$ years. Details can be found in [7], available from [18].

Several improvements have been made to MoonWalker 1.0 to enhance its usability, including a user-friendly error tracer, an extensive test framework, and an implementation of structured exception handling. Furthermore, two partial order reduction (POR) [4] techniques have been implemented into MoonWalker 1.0: (i) POR using object escape analysis (which is also used by JPF) and (ii) stateful dynamic POR. Finally, two novel techniques have been implemented in MoonWalker 1.0, which will be discussed in more detail below.

Memoised Garbage Collector. MoonWalker 0.5 used the well known Mark & Sweep algorithm (M&S) for garbage collection (GC). A drawback of M&S is that it is global: for each invocation (i.e. after each transition), the whole heap is visited twice to remove dead objects. In other words, M&S cannot exploit the locality of transitions within a (software) model checker.

For MoonWalker 1.0 we took a different approach, which is based on an incremental shortest-path algorithm for single-source directed graphs with positive weights [9]. We devised and implemented the Memoised Garbage Collector (MGC) algorithm, which uses information retrieved from changes between successive states to determine which objects should be garbage collected.

The basic idea is to track for each object in the heap its depth from the root elements (on the call stacks of the threads). Upon changes to the heap, the tracked depth of the changed objects become inconsistent, and their depths need to be recalculated. When the changes to the heap are small – which they usually are – only a small part of the heap needs to be traversed. If the depth of an object becomes infinity, we know that the object has become unreachable.

We know of one other software model checker which uses a non-global garbage collector: JNUKE [2]. JNUKE uses a generational garbage collection (GGC) as described in [3]. Although the objectives of both non-global GC algorithms are the same, the implementations are substantially different. MGC is provable precise [7], whereas GGC is not, because the latter exploits the heuristic that only new objects are likely to be garbage collected. For MoonWalker unpreciseness is undesired because this may cause the state matcher to determine that two semantically equivalent states are different [6].

MGC has a better time-complexity than M&S, which is the dominant garbage collection in use by software model checkers. Experiments showed that the use of MGC increases performance of about 10-25%, depending on the model and its state space. Details on MGC can be found in [7,8].

Collapsing Interleaving Information. The dynamic POR algorithm by Flanagan & Godefroid [4] only works correctly for *stateless* exploration. The issue lies in

the correct dynamic POR semantics upon a state revisit. A naive and incorrect stateful adaptation of dynamic POR would backtrack upon exploration of a revisited state. This is incorrect, because mutual dependencies between transitions in the state space below the revisited state and the current path to the revisited state would not be considered. This leads to over-aggressive reduction. Both [10] and [13] independently observed this, and proposed similar solutions. The idea is to mimic a stateless search upon a revisit by recalling all necessary *interleaving information* about the state space below the revisited state and inject the appropriate transitions in the working sets on the current DFS stack.

[10,13] observe that stateful dynamic POR uses a lot of memory and suggest (as future work) to compress the interleaving information used for stateful dynamic POR. MoonWalker 1.0 improves upon [13] by compressing the interleaving information by canonicalisation followed by collapse compression. The collapse compression step exploits the notion that the interleaving information of states do not change much between successive states. We reuse the structered state collapsion scheme that was already present in MoonWalker 0.5.

Experiments show (again depending on the model and the state space) that dynamic POR may reduce the memory consumption by a factor of two.

Implementation. The current version of MoonWalker is version 1.0. The total development of the tool took roughly two man years of work. The code base of MoonWalker 1.0 consists of 17k lines of C# code and constitutes 475Kb of source code. Both a binary and source distribution are available from [18].

MoonWalker 1.0 supports 74 CIL bytecode instructions. These are all possible instructions that can be emitted by MONO's C# 1.x compiler. Support for the last nine missing instructions is future work.

Experiments. Apart from using small experiments that synthesise a small scenario, we also used the *Java Grande Forum Benchmarks (JGF)* [14] for evaluating MoonWalker against JPF. JGF is a mature benchmark suite developed for the scientific community, which contains real life examples. Of the three multi-threaded parallel benchmarks within this suite, we used the *MolDyn* benchmark (loc: 965, size: 26Kb) and *Raytracer* benchmark (loc: 1540, size: 49Kb). We ported these two benchmarks to C# for use with MoonWalker.

Results show that MoonWalker and JPF are on par in terms of performance. Differences between the two tools are small. Both tools typically explore about 1000-5000 states/sec. MoonWalker is faster in terms of states per second, but JPF is better at reducing the state space because its POR object escape analysis algorithm also uses locking information. The results also indicate that MoonWalker utilises memory relatively less efficiently than JPF. This is caused by the memory overhead incurred by stateful dynamic POR. Details on the experiments can be found in chapter 5 of [7].

3   Conclusions

In this paper we presented MoonWalker 1.0, a model-checker for CIL bytecode programs. Due to several refactorings, the design and implementation of

MoonWalker is clear, readable and extensible. We feel that MoonWalker is a useful platform in an academic environment where ease of experimentation with different implementations is an important virtue. To extend the usability of MoonWalker further, several improvements are planned:

- Further improvements to POR and state compression;
- Optimisations to the (memoised) garbage collector;
- Mixing of symbolic and concrete data;
- Multi-threaded and distributed version of MoonWalker;
- Case studies with other programming languages than C#;
- Support for C# 3.0 and .Net 3.5.

References

1. Aan de Brugh, N.H.M.: Software Model Checking for Mono. Master's thesis, University of Twente, Enschede, The Netherlands (August 2006)
2. Artho, C., Schuppan, V., Biere, A., Eugster, P., Baur, M., Zweimüller, B.: JNuke: Efficient Dynamic Analysis for Java. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 462–465. Springer, Heidelberg (2004)
3. Fargas, P.: Garbage Collection for JNuke. Master's thesis, ETH Zürich, Switzerland (September 2004)
4. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model Checking Software. In: Proc. of POPL 2005, pp. 110–121. ACM Press, New York (2005)
5. Grieskamp, W., Tillmann, N., Schulte, W.: XRT: Exploring Runtime for. NET - Architecture and Applications. ENTCS 144(3), 3–26 (2006); Proc. of SoftMC 2005
6. Iosif, R., Sisto, R.: Using Garbage Collection in Model Checking. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 20–33. Springer, Heidelberg (2000)
7. Nguyen, V.Y.: Optimising Techniques for Model Checkers. Master's thesis, University of Twente, Enschede, The Netherlands (December 2007)
8. Nguyen, V.Y., Ruys, T.C.: Memoised Garbage Collection for Software Model Checking. In: Knowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 201–214. Springer, Heidelberg (2009)
9. Ramalingam, G., Reps, T.W.: An Incremental Algorithm for a Generalization of the Shortest-Path Problem. Journal Algorithms 21(2), 267–305 (1996)
10. Ranganath, V.P., Hatcliff, J., Robby.: Enabling Efficient Partial Order Reductions for Model Checking Object-Oriented Programs. Technical Report SAnToS-TR2007-2, SAnToS Laboratory, CIS Department, Kansas State University (2007)
11. Ruys, T.C., Aan de Brugh, N.H.M.: MMC: the Mono Model Checker. ENTCS 190(1), 149–160 (2007); Proc. of Bytecode 2007, Braga, Portugal
12. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. ASE 10(2), 203–232 (2003)
13. Yi, X., Wang, J., Yang, X.: Stateful Dynamic Partial-Order Reduction. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 149–167. Springer, Heidelberg (2006)
14. The Java Grande Forum Benchmark Suite,
    `http://www.epcc.ed.ac.uk/research/activities/java-grande/`
15. Java PathFinder, `http://javapathfinder.sourceforge.net/`
16. The Mono Project, `http://www.mono-project.com/`
17. .NET Languages, `http://www.dotnetlanguages.net/`
18. Moonwalker, `http://www.cs.utwente.nl/~ruys/moonwalker/`

Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays

Robert Brummayer and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria
{robert.brummayer,armin.biere}@jku.at

Abstract. Satisfiability Modulo Theories (SMT) is the problem of deciding satisfiability of a logical formula, expressed in a combination of first-order theories. We present the architecture and selected features of Boolector, which is an efficient SMT solver for the quantifier-free theories of bit-vectors and arrays. It uses term rewriting, bit-blasting to handle bit-vectors, and lemmas on demand for arrays.

1 Introduction

A new kind of verification engines, called Satisfiability Modulo Theories (SMT) solvers, gained a lot of interest both in research and industry recently. SMT generalizes pure boolean satisfiability (SAT) and provides first-order theories to express design and verification conditions of interest. For example, important first-order theories are fixed-size bit-vectors, arrays, linear arithmetic, and difference logic. An SMT solver takes a formula expressed in a combination of first-order theories as input, and decides satisfiability. Additionally, if the instance is satisfiable, then most SMT solvers provide a model. For more information about SMT and first-order theories see for example [4,10,12].

Boolector is an efficient SMT solver for the quantifier-free theory of bit-vectors in combination with the extensional theory of arrays. Bit-vectors can be used to express designs and specifications directly on the word-level, while arrays can be used to model memory, e.g. main memory in software, or memory components like caches and FIFOs in hardware systems. The combination of bit-vectors and arrays allows reasoning about software and hardware on a more appropriate level than pure propositonal logic. Generally, SMT solvers benefit from the additional word-level information and generates word-level models. Boolector provides concrete models for bit-vector *and* arrays.

The SMT competition [2] in 2008 showed, that there has been a lot of progress in the SMT community. In each division the winner clearly outperformed last year's winner. In particular, the performance of SMT solvers in the quantifier-free theory of bit-vectors `QF_BV`, and bit-vectors with arrays and uninterpreted functions `QF_AUFBV`, increased heavily. Boolector entered the SMT competition for the first time. It participated in exactly these two divisions and won both. In `QF_BV` it solved 18 formulas more than Z3.2 and 92 formulas more than last

year's winner Spear v1.9 [1]. In `QF_AUFBV` Boolector solved 16 formulas more than Z3.2 and 64 more than last year's winner Z3 0.1 [8].

2 Architecture

Boolector depends on term rewriting and bit-blasting for bit-vectors. Lemmas on demand [9] are used to handle the extensional theory of arrays lazily [5]. It takes a formula expressed in the SMT-LIB format [11], or alternatively in the BTOR format [6], as input. The BTOR format is a low-level bit-vector format with clean semantics that is easy to parse. Additionally, BTOR supports bit-vector arrays and model checking of safety properties. In addition to these input formats, Boolector also provides a rich C API, which allows to use Boolector as library. Boolector is implemented in C. A schematic overview is shown in Fig. 1.
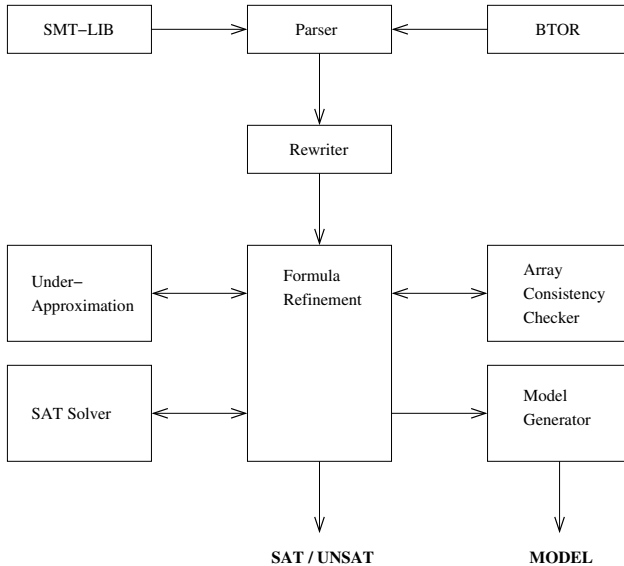
Fig. 1. Schematic overview of Boolector

Parser. The parser reads the input and builds an abstract syntax DAG. During the parse process, basic rewriting rules are applied to simplify the DAG.

Rewriter. The rewriting engine provides rewrite rules that can be divided into three levels. By default, all rewrite levels are applied to simplify the input. Level 1 rewrite rules are basic rules, e.g. $a \wedge \neg a \Leftrightarrow \bot$, that are applied during formula construction. Level 2 rewrite rules consist of global term substitutions in combination with static analysis techniques. Before terms are substituted, they are topologically sorted. Then, term substitutions are performed in one pass. Level 3

rewrite rules perform arithmetic normalization. Rewrite rules of quadratic worst case complexity are additionally bounded in recursion depth.

Array Consistency Checker. This checker is one main component in the lemmas on demand approach for the extensional theory of arrays [5]. It checks if the current assignment by the SAT solver is consistent with the theory.

Under-approximation. Boolector supports under-approximation of bit-vector variables and reads on arrays. This module is responsible for realizing under-approximation directly on the CNF level. Under-approximation constraints are added as clauses that additionally constrain the search space.

SAT Solver. PicoSAT [3] is used as SAT solver. It uses state of the art techniques like watching literals, phase saving, conflict learning and rapid restarts. Boolector uses PicoSAT incrementally to decide the extensional theory of arrays, and for under-approximation.

Model Generator. The ability to provide concrete models in the satisfiable case cannot be overestimated. In generaral, a satisfiable formula corresponds to a bug that has been found. The ability to provide a concrete counter example that can be directly used for debugging is one of the main features in the success story of model checking [7]. Boolector can output concrete bit-vector *and* array models. The array models are (partially) instantiated arrays, where indices and values are concrete bit-vectors.

Formula Refinement. The formula refinement is the heart of Boolector. Initially, the bit-vector part is translated to SAT while the array part is abstracted with the help of fresh bit-vector variables. The refinement loop calls the SAT solver to obtain an assignment. If the result is unsatisfiable, the loop terminates with *unsatisfiable*. However, if the result is satisfiable, the array consistency checker is used to check if the current assignment is consistent with the extensional theory of arrays. If the current assignment is consistent, then the formula is *satisfiable*. However, if the assignment is inconsistent, a lemma on demand, that rules out this and similar assignments, is added as as formula refinement. Additionally, under-approximation refinement can be enabled for bit-vector variables and reads. In this case the under-approximation refinement and the lemmas on demand refinement for the extensional theory of arrays are intertwined.

3   Selected Features

Model Checking. Boolector can also be used as incremental model checker for word-level safety properties of synchronous hardware systems with memories [6]. The BTOR format provides a sequential "next" operator, which can be used to express state transitions of bit-vector registers and memories. Boolector supports bounded model checking for witnesses, and k-induction with and without all-different constraints. All-different constraints are used for simple paths.

Under-approximation. Boolector supports several under-approximation techniques and refinement strategies. Bit-vector variables and reads on arrays can

be additionally constrained on the CNF level. The under-approximation can be refined locally or globally. In the global refinement strategy, the under-approximation is refined equally for all variables and reads. In the local strategy the under-approximation refinement is individually performed for each variable and read. The under-approximation feature allows to generate "smaller" models that are typically easier to interpret by users.

Pretty Printer. Boolector allows to convert formulas from BTOR to SMT-LIB format and vice versa. The pretty printer can be combined with Boolector's rewriting module to internally simplify the formula before conversion.

4 Conclusion

We presented Boolector, which is an efficient SMT solver for the quantifier-free theories of bit-vectors and arrays. Boolector uses term rewriting, bit-blasting for bit-vectors, and lemmas on demand for arrays. We discussed its architecture, main concepts, and selected features.

References

1. Babic, D.: Exploiting Structure for Scalable Software Verification. PhD thesis (2008)
2. Barrett, C., Deters, M., Oliveras, A., Stump, A.: SMT-Comp. (2008), `http://www.smtcomp.org`
3. Biere, A.: PicoSAT essentials. JSAT 4 (2008)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Heidelberg (2007)
5. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. In: Proc. SMT (2008)
6. Brummayer, R., Biere, A., Lonsing, F.: BTOR: Bit-precise modelling of word-level problems for model checking. In: Proc. BPR (2008)
7. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008)
8. 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)
9. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Proc. SAT (2002)
10. Kroening, D., Strichman, O.: Decision Procedures: An algorithmic Point of View. Springer, Heidelberg (2008)
11. Ranise, S., Tinelli, C.: The satisfiability modulo theories library, SMT-LIB (2008), `http://www.smt-lib.org`
12. Sebastiani, R.: Lazy satisfiability modulo theories. JSAT, 3 (2007)

The Yogi Project: Software Property Checking via Static Analysis and Testing

Aditya V. Nori[1], Sriram K. Rajamani[1], SaiDeep Tetali[1],
and Aditya V. Thakur[2]

[1] Microsoft Research India
{adityan,sriram,v-saitet}@microsoft.com
[2] University of Wisconsin-Madison
adi@cs.wisc.edu

Abstract. We present Yogi, a tool that checks properties of C programs by combining static analysis and testing. Yogi implements the Dash algorithm which performs verification by combining directed testing and abstraction. We have engineered Yogi in such a way that it plugs into Microsoft's Static Driver Verifier framework. We have used this framework to run Yogi on 69 Windows Vista drivers with 85 properties. We find that the new algorithm enables Yogi to scale much better than Slam, which is the current engine driving Microsoft's Static Driver Verifier.

1 Introduction

Static analysis and testing have always had complementary strengths and weaknesses. With static analysis, we can obtain very good coverage and analyze program paths that are hard to exercise using testing, but we are forced to deal with scalability issues and false errors. With runtime testing, we can obtain only partial coverage, but the approach scales to large programs and every error that is reported is indeed realizable. Thus, attempting to combine the complementary strengths of static analysis and runtime testing is natural.

For the past few years, we have been investigating methods for combining static analysis in the style of counter-example driven refinement ala Slam [1], with runtime testing and automatic test case generation approaches in the style of concolic execution ala Dart [5]. Our first attempt in this direction was the Synergy algorithm [6], which handled single procedure programs with only integer variables. Then, we proposed Dash [3], which had new ideas to handle pointer aliasing and procedure calls in programs. Throughout this evolution, Yogi has been our implementation vehicle to realize and evaluate these algorithms. Currently, Yogi implements the Dash algorithm. We have spent over 3 person-years of engineering to make the tool robust and usable – Yogi has been run over several hundreds of thousands of lines of C code, with several properties.

We describe the design and engineering of Yogi in this paper. The Synergy and Dash algorithms themselves are described in [6,3]. Section 2 outlines the

(a) Architecture of Yogi

```
state {
    enum { Unlocked=0, Locked=1 }
        state = Unlocked;
}

KeAcquireSpinLock.return {
    if (state == Locked)
        abort;
    else
        state = Locked;
}

KeReleaseSpinLock.return {
    if (state == Unlocked)
        abort;
    else
        state = Unlocked;
}
```

(b) An example Slic specification
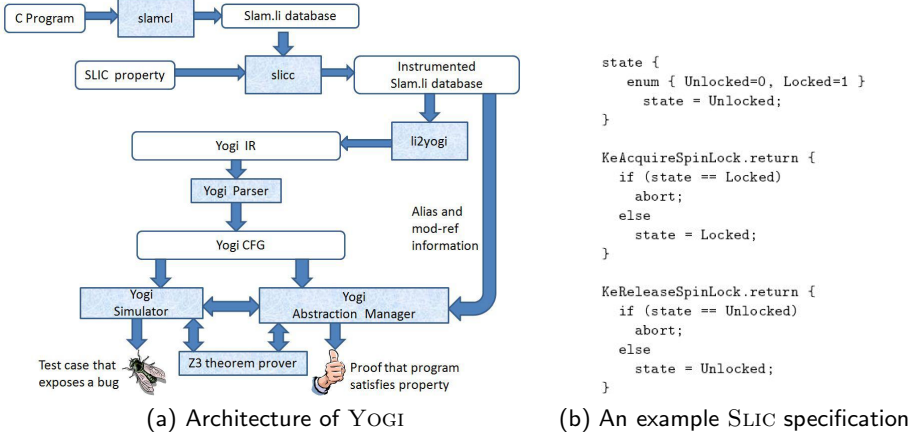
Fig. 1.

architecture and various components in Yogi. Section 3 describes empirical results from running Yogi on 69 Windows Vista device drivers with 85 properties and Section 4 concludes the paper by discussing current status of Yogi.

2   Architecture

As shown in Figure 1(a), Yogi takes two inputs: (1) a C program, and (2) a safety property specified in the Slic specification language [2]. A sample Slic specification for a locking protocol (KeAcquireSpinLock and KeReleaseSpinLock occur in strict alternation) is shown in Figure 1(b). Yogi uses Slam's front-end (called slamcl) to parse C programs and Slam's property instrumentor (called slicc) to instrument the property into the program. The resulting program with the property instrumented is in Slam's internal binary format called li. We have developed a translator called li2yogi that converts the li format to Yogi's intermediate form called yogi-ir. The yogi-ir is a textual format that represents the program at the level of basic blocks with instructions. Each instruction is one of three types: an assignment, assume statement or a procedure call. Once a program has been converted to the yogi-ir format, it is read by the YogiParser to produce an internal inter-procedural control flow graph.

The two main components of Yogi are: (1) Ysim, a simulator which can perform both concrete execution with concrete values and symbolic execution, and (2) YabsMan, an abstraction manager, which manages proofs.

The Ysim simulator code is polymorphic over the type of the values it operates. Thus, the same simulation code does both concrete and symbolic execution. During concrete execution, the simulator uses a model of memory where concrete values of appropriate type are stored in locations. During symbolic execution, the simulator stores symbols and formulas in locations. It uses the Z3 theorem prover [4] to reason about consistency of the formulas and to generate test cases as satisfiable models of formulas. The YabsMan abstraction manager maintains

a region graph abstraction of the program. For each control point in a program YABSMAN maintains a finite partition over the set of states. Each partition is represented by a predicate, which is a Z3 formula.

YOGI implements the DASH algorithm [3]. The DASH algorithm simultaneously maintains a forest of test runs and a region-graph abstraction of the program. Tests are used to find bugs and abstractions are used to prove their absence. During every iteration, if a concrete test has managed to reach the error region, a bug has been found. If no path in the abstract region graph exists from the initial region to the error region, a proof of correctness has been found. If neither of the above two cases are true, then we have an abstract counterexample, which is a sequence of regions in the abstract region graph, along which a test can be potentially driven to reveal a bug. The DASH algorithm crucially relies on the notion of a *frontier* [6,3], which is the boundary between tested and untested regions along an abstract counterexample that a concrete test has managed to reach. In every iteration, the algorithm first attempts to extend the frontier using test case generation techniques similar to DART. If test case generation fails, then the algorithm refines the abstract region graph so as to eliminate the abstract counterexample.

YOGI performs modular verification. For function calls, YOGI uses an initial abstraction that is based on locations that the procedure modifies. A conservative alias analysis is used to get an overapproximation to the set of locations modified by the procedure and this is used to build an initial summary for each function. If a procedure call occurs at the frontier, then the summary so computed is first used to see if a refinement can rule out the abstract counterexample. If this is not possible, YOGI tries to generate a test case through the procedure, and see if the test case extends the frontier. If the test case so generated does not extend the frontier, then YOGI descends into the called procedure and analyzes the procedure in detail [3].

3   Empirical Results

We have integrated YOGI with Microsoft's Static Driver Verifier framework. We have tested YOGI with the Static Driver Verifer's integration test pass suite, which contains 69 device drivers and 85 properties, a total of 5865 driver-property pairs. The largest driver in this pass has over 30K lines of code, and the total size of all the drivers is over 300K lines of code.

At the time of this writing YOGI finishes on 95% of the runs on the integration test pass. It is able to both prove properties correct and find bugs in the driver code. In comparison with SLAM there are 129 runs where SLAM either times out or spaces out, where YOGI is able to give a result. The total time taken by YOGI to run over all the 5865 runs is about 32 hours on an 4 core machine, compared to over 69 hours taken by SLAM.

A comparison of YOGI with SLAM on 16 representative drivers is shown in Table 1. Every row of this table shows the driver, its number of lines of code, the number of properties checked and the time (in minutes) taken by SLAM and YOGI along with the number of time-outs (set to 30 minutes).

Table 1. Empirical evaluation of Yogi on 16 device drivers

Program	Lines	Properties	Slam		Yogi	
			Time-outs	Time (min)	Time-outs	Time (min)
parport	34196	19	1	**91.2**	0	**26.1**
serial1	32385	21	3	**142.4**	0	**21.5**
serial	31861	21	3	**203.9**	0	**28.1**
fdc_fail	9251	50	0	**117.6**	0	**8**
kbdclass1	7426	38	2	**124.9**	0	**115**
kbdclass	7132	36	2	**125.5**	0	**90.4**
serenum	6011	38	1	**95.6**	0	**10.9**
pscr	5680	37	0	**55**	0	**26.4**
modem	3467	19	0	**18**	0	**22.3**
1394Vdev	2757	22	2	**90.7**	0	**72.9**
1394Diag	2745	23	3	**121.4**	0	**68.8**
diskperf	2351	31	0	**36.8**	1	**100**
incomplete1	1558	29	0	**16**	0	**6.3**
toastmon1	1539	32	0	**13.5**	0	**8.4**
toastmon	1505	32	0	**16.6**	0	**7.6**
daytona	565	29	1	**106.9**	0	**77.4**

4   Current Status

Yogi is a stable and robust tool that has been run over several hundreds of thousands of lines of C code. At the tool demonstration, we will show Yogi running on small programs and demonstrate its ability to find bugs and prove programs correct. We will also present the results from running Yogi on the 5865 runs from Static Driver Verifier's integration test pass (a subset of these results are shown in Table 1).

References

1. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
2. Ball, T., Rajamani, S.K.: Slic: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research (2001)
3. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA 2008: International Symposium on Software Testing and Analysis, pp. 103–122. ACM Press, New York (2008)
4. de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: PLDI 2005: Programming Language Design and Implementation, pp. 213–223. ACM Press, New York (2005)
6. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: FSE 2006: Foundations of Software Engineering, pp. 117–127. ACM Press, New York (2006)

TaPAS: The Talence Presburger Arithmetic Suite

Jérôme Leroux and Gérald Point

LaBRI, Université Bordeaux 1, CNRS
{leroux,point}@labri.fr

Abstract. TAPAS is a suite of libraries dedicated to FO $(\mathbb{R}, \mathbb{Z}, +, \leq)$. The suite provides (1) the application programming interface GENEPI for this logic with encapsulations of many classical solvers, (2) the BDD-like library SATAF used for encoding Presburger formulae to automata, and (3) the very first implementation of an algorithm decoding automata to Presburger formulae.

1 The Mixed Additive Theory

The automatic verification of reactive systems is a major field of research. These systems are usually modeled with variables taking values in some infinite domains. Popular approaches for analyzing these models are based on decision procedures adapted to the variables domains. For numerical variables the mixed additive theory FO $(\mathbb{R}, \mathbb{Z}, +, \leq)$ provides a natural logic to express linear constraints between integral variables and real variables. This logic has positive aspects: it is decidable and actually many solvers implement decision procedures for the full logic or some sub-logics like the Presburger arithmetic. TAPAS is a suite of libraries dedicated to these logics.

The sequel is organized as follows. In Section 2 the architecture of TAPAS is presented. Section 3 presents the BDD-like library SATAF used for encoding Presburger formulae to automata and for decoding automata to Presburger formulae. Finally, Section 4 provides some benchmarks.

2 TAPAS at a Glance

The following figure shows the architecture of TAPAS (enclosed by the dotted line). The FAST model-checker is also depicted but it does not actually belong to the suite; it is just a client application.
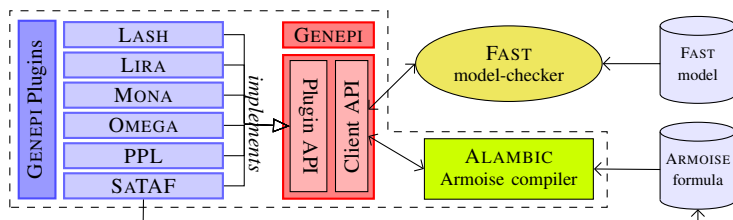
Selecting the most efficient solver for a given application can be difficult. This choice can change dramatically the practical performance of the application. Since solvers have

incompatible application programming interfaces (API), a particular one must be first selected prior the implementation of the application; change afterward require additional development effort. The GENEPI library[BLP06] addresses this issue by offering small APIs between, on one side, applications requiring solvers and, on the other side, solvers implementing decision procedures. The connection between applications and solvers is realized transparently using dynamically loaded modules (*plugins*) specified at the execution time. This way the choice of a solver can be postponed after the implementation of an application based on GENEPI; the best one can be selected by performing benchmarks. Since [BLP06], GENEPI has been enhanced with new features: support for $\mathbb{R}$, new plugins (LIRA and PPL), . . .

In general, specifying sets with low level logics is difficult and fastidious. The same problem appears with FO $(\mathbb{R}, \mathbb{Z}, +, \leq)$. The ARMOISE language allows to describe concisely sets of vectors (in $\mathbb{Z}$ and/or $\mathbb{R}$). The succinctness of formulae is achieved using, among other tricks, hierarchical definitions and arithmetic over sets (which hides numerous and ugly quantifications). Below is an example of an ARMOISE formula which specifies a repeated pattern in $\mathbb{N}^2$. Note that ARMOISE formulae may define sets which are not necessarily definable in FO $(\mathbb{R}, \mathbb{Z}, +, \leq)$.

```
let
    origin := (7,3);
    directions := { (4,0), (0,4), (4,4) };
    pattern := ([0...2], [0...2]) \ (1,1);
in
    origin + nat * directions + pattern;
```

TAPAS provides tools to support the ARMOISE language. The ALAMBIC library permits to compile ARMOISE formulae into calls to GENEPI functions. An important step in this compilation process is the translation of high-level constructions into the mixed additive theory; due to the expressive power of ARMOISE this transformation is not always possible.

3   SATAF: Shared-Automata and the Synthesis of Formulae

Many solvers like LASH, LIRA and MONA are based on automata packages. Intuitively, mappings from words to numerical vectors are used to associate to automata potentially infinite sets of numerical vectors (one vector per accepted word). Usually, the minimal deterministic automata are proved canonically associated to the represented sets and not on the way they have been computed. In particular these solvers are well adapted to sets obtained after long chains of operations like in symbolic model checking. In [Cou04], Couvreur introduced a new data-structure called *shared-automata*. Intuitively, the sharing of a finite set of automata is performed by merging every states equivalent with respect to the Nérode relation. TAPAS contains the SATAF library that implements this data structure. In SATAF, deciding if two automata recognize the same language reduces to check the equality of their pointers. Similarly to BDD packages, SATAF uses a pointer-based hash-table cache algorithm. Up to our knowledge, no other automata package implements these features. Since the first version of SATAF written in JAVA by Couvreur [Cou04], a new version written in C is now maintained in TAPAS. TAPAS also provides an implementation of the GENEPI API based on SATAF.

Extracting geometrical properties (for instance linear constraints) from automata is a challenging problem. From a theoretical point of view, this problem has been solved in [Ler05]. This article provides a polynomial time algorithm for computing Presburger formulae from automata representing Presburger sets. The algorithm first extracts the "necessary" linear constraints from the automaton. Then, it computes an unquantified Presburger formula using only these constraints, Boolean operations, translations by integer vectors, and scaling by integer values. An implementation of this algorithm is provided with SATAF. The generated formulae follows the ARMOISE syntax. Note that SATAF, GENEPI and ALAMBIC provide together the very first implementation of an algorithm that normalizes Presburger formulae into unique canonical forms that only depend on the denoted sets. Intuitively in the normalization process, the minimization procedure for automata acts like a simplification procedure for the Presburger arithmetic.

4   Experimenting the Automata to Formulae Algorithm

FAST [BLP06] is a tool for verifying reachability properties of infinite-state systems. The tool is implemented over TAPAS. Benchmarks of FAST on various GENEPI implementations are available in [BLP06, BDEK07]. We experimented the computation of ARMOISE specifications from SATAF automata denoting the reachability sets of 40 systems. Some results are presented in the following table. Column "$\rightarrow \mathcal{A}$" is the time in seconds spent for computing an automaton $\mathcal{A}$ representing the reachability set, "$|\mathcal{A}|$" is the number of states of $\mathcal{A}$, "$\mathcal{A} \rightarrow S$" is the time in seconds to produce an ARMOISE specification S from $\mathcal{A}$, $|S|$ is the number of characters of S, "n" is the number of variables of S, and "l" is the number of linear constraints generated in S.

system	$\rightarrow \mathcal{A}$	$|\mathcal{A}|$	$\mathcal{A} \rightarrow S$	$|S|$	n	l
Central Server system	9.57	75	0.07	3716	12	10
Consistency Protocol	194.96	90	0.06	3811	11	11
CSM - N	24.38	66	0.05	3330	14	11
Dekker ME	17.73	200	0.01	13811	22	0
Multipoll	11.38	612	2.54	60995	18	19
SWIMMING POOL	71.03	553	0.14	1803	9	18
Time-Triggered Protocol	116.89	17971	90.53	984047	11	44

We also experimented the normalization of ARMOISE formulae on various examples. Due to space limitation, only four representative examples are presented: `modulo`, `large-coef`, `large-var` and `unsimplified`. Example `modulo` denotes the Cartesian product $(11\mathbb{N}) \times (7\mathbb{N}) \times (5\mathbb{N}) \times (3\mathbb{N})$ where $m\mathbb{N}$ is the set of non-negative integers multiple of m, `large-coef` is the conjunction of three linear constraints with coefficients larger than 20, `large-var` is a linear constraint defined over 36 variables, and `unsimplified` is a disjunction of two sets of linear constraints with redundant constraints. Benchmark results are summarized in the following table. Columns have the following meaning: "example" provides the name of the ARMOISE specification S_0, "$|S_0|$" is the number of characters of S_0, "$S_0 \rightarrow \mathcal{A}$" is the time in seconds to produce a SATAF automaton $\mathcal{A}$ from S_0. The other columns have been defined previously.

example	$|S_0|$	$S_0 \rightarrow \mathcal{A}$	$|\mathcal{A}|$	$\mathcal{A} \rightarrow S$	$|S|$	n	l
modulo	40	0.1	4620	35.8	335	4	0
large-coef	167	1.7	147378	25.1	957	4	3
large-var	543	0.2	4320	12.7	2220	36	1
unsimplified	529	1.1	16530	1.3	1026	5	2

We observe that the computation of SATAF automata $\mathcal{A}$ from ARMOISE specifications S_0 takes less than 2 seconds. Moreover, the computation of ARMOISE specifications S from $\mathcal{A}$ takes less than half a minute even on automata with more than 10^5 states. In practice the implementation (quadratically) slows down in presence of modular constraints. Note that $|S_0| < |S|$ in all our examples due to non-optimized outputs. However, even if $|S_0| < |S|$ in the last example `unsimplified`, the redundant linear constraints of S_0 no longer appear in S.

Contrary to previous results, we observe that $|S|$ is quite smaller than $|\mathcal{A}|$. Recall that the generated ARMOISE specifications are combinations of linear constraints. In both series of benchmarks formulae contain a few number of constraints (see the "l" column). The complexity of the combinations explains the differences in the results. In practice, we observe that small automata can encode complex combinations but only a small number of linear constraints.

5 Conclusion and Future Work

TAPAS is distributed at `http://altarica.labri.fr/wiki/tools:tapas:` under GPLv2. Thanks to GENEPI, this is the first tool to our knowledge which offers (1) a simple framework for the development of applications requiring solvers for the mixed additive theory and (2) an easy way to benchmark solvers. TAPAS also provides the SATAF solver based on shared-automata and the first implementation of the algorithm translating them into equivalent ARMOISE formulae.

Future Work. Predicate abstraction methods are limited by the number of considered predicates. Usually some of these predicates are redundant. More precisely, some predicates are not semantically used even if they syntactically appear. In future work, we are interested in using the normalization procedure of TAPAS in order to remove redundant predicates. In fact, the set of extracted linear constraints from an automaton is known minimal : these constraints syntactically appears in any unquantified Presburger formula that denotes the set represented by the automaton.

References

[BDEK07]   Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: LIRA: Handling constraints of linear arithmetics over the integers and the reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 307–310. Springer, Heidelberg (2007)

[BLP06]   Bardin, S., Leroux, J., Point, G.: FAST extended release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)

[Cou04]   Couvreur, J.-M.: A BDD-like implementation of an automata package. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 310–311. Springer, Heidelberg (2005)

[Ler05]   Leroux, J.: A polynomial time presburger criterion and synthesis for number decision diagrams. In: LICS, pp. 147–156. IEEE Comp. Soc., Los Alamitos (2005)

Transition-Based Directed Model Checking

Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski

University of Freiburg
Department of Computer Science
Freiburg, Germany
{mwehrle, kupfersc, podelski}@informatik.uni-freiburg.de

Abstract. Directed model checking is a well-established technique that is tailored to fast detection of system states that violate a given safety property. This is achieved by influencing the order in which states are explored during the state space traversal. The order is typically determined by an abstract distance function that estimates a state's distance to a nearest error state. In this paper, we propose a general enhancement to directed model checking based on the evaluation of *state transitions*. We present a schema, parametrized by an abstract distance function, to evaluate transitions and propose a new method for the state space traversal. Our framework can be applied automatically to a wide range of abstract distance functions. The empirical evaluation impressively shows its practical potential. Apparently, the new method identifies a sweet spot in the trade-off between scalability (memory consumption) and short error traces.

1 Introduction

When model checking safety properties, the ultimate goal is to prove the absence of error states. This can be done by exploring the entire reachable state space. However, the state space of realistic applications is often too large to be enumerated exhaustively because of the state explosion problem. Directed model checking is a well-established technique to tackle this problem and has found its way in state-of-the-art tools such as SPIN, PATHFINDER or UPPAAL [5, 6, 11]. In directed model checking, the state space traversal is guided ("directed") based on specific criteria towards error states. Generally, these guidance criteria are automatically extracted from the model by taking an abstraction of the model and computing an abstract distance function $d^{\#}$. For a state s, the value $d^{\#}(s)$ approximates the distance of s to a nearest error state. These values are used during the state space traversal in order to determine which state is explored next.

Each different version of directed model checking thus arises through the choice of the abstraction that is used to compute the abstract distance function, and by the choice of the basic (non-deterministic) algorithm for traversing the state space. Earlier work on directed model checking was mainly focused on the first point, i. e., in defining abstractions that lead to distance estimation functions $d^{\#}$ to guide the state space traversal efficiently towards an error state [3, 4, 5, 9, 12, 13, 17, 18]. Considering the second point, there are two predominantly used algorithms of directed model checking, namely A* and *greedy search* (cf. [16]). A* is guaranteed to find shortest possible error traces for certain kinds of distance estimation functions, but is often too memory consuming

for large systems. Greedy search does not necessarily find shortest possible error traces, but mostly scales much better than A* in practice.

In this paper, we present a new version of directed model checking that seems to identify a sweet spot in the trade-off between scalability (memory consumption) and short computed error traces. It is based on the concept of *useless transitions* which is an adaptation of the *useless actions* approach that has been introduced in the context of AI Planning [20]. As indicated by its name, the concept of useless transitions extends directed model checking by additionally evaluating transitions, not just states. We will see that this is a general concept in the sense that useless transitions can be computed fully automatically with the given distance estimation function $d^\#$. That is, whatever the choice of the underlying abstraction for computing the abstract distance function has been, we can use the already computed abstract distance function in order to effectively recognize useless transitions. We will characterize a class of distance estimation functions for which our method is suited best. We define a new (non-deterministic) strategy for state space traversal that takes these useless transitions into account. The new strategy is an amalgam of the two strategies A* and greedy search. For the two extreme cases of abstraction, it becomes the former or the latter, respectively.

We have implemented our method and have applied it to a number of academic and industrial benchmarks. This allowed us to experimentally compare the new directed model checking method with the two existing predominant methods A* and greedy search. The empirical results impressively show the benefit of our approach: We obtain almost shortest error traces, whereas the number of expanded states reduces significantly compared to A* and also to greedy search in most cases.

We will next give an example that greatly oversimplifies the issues at hand but gives an intuition about useless transitions and their potential usefulness.

Example. Figure 1 depicts a system consisting of $n + 1$ parallel components A_0 to A_n. The initial state of the system is $(s_0^0, s_0^1, \ldots, s_0^n)$ and the error state is $(s_1^0, s_1^1, \ldots, s_1^n)$. Suppose that we apply directed model checking to check if this error state is reachable. Further suppose that we therefore use the *maximum graph distance* as the abstract distance function [4, 5]. This function is based on the local distance of each single automaton A_i. More precisely, let $d(i)$ denote the graph distance from A_i's current location to A_i's error state, then the maximum graph distance is defined as $\max_{i=0, \ldots, n} d(i)$.

It turns out that, for this problem, the maximum graph distance is a rather uninformed distance function. It cannot distinguish states that are nearer to the error state from others. If there is at least one local state of the form s_0^i, then the abstract distance value is 1. We may characterize the state space topology induced by this abstract distance function as follows. There is one single plateau (for all of the 2^{n+1} reachable system states but for the error state the abstract distance is 1). This means that the guidance based on this abstract distance function is very poor. In fact, for every abstract distance function, the similar situation arises.

In the example, it is trivial to see that each state transition where a local state leaves a local error state (depicted by a double circle) should be avoided as much as possible during the state space traversal (it is a useless transition!). Without steps corresponding to such transitions, the state space traversal stops after $n + 1$ steps and returns the shortest possible error path.
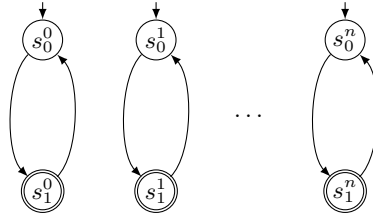
Fig. 1. An automata system with $n + 1$ components

The remainder of this paper is organized as follows. In the next section, we discuss related work. In Section 3, we give the preliminaries needed for this work, including a more detailed introduction to directed model checking. In Section 4, we introduce the notion of useless transitions and present our directed model checking algorithm based on this concept. In Section 5, we empirically evaluate our algorithm on a number of benchmarks. Section 6 concludes the paper.

2   Related Work

Research on directed model checking so far mainly focused on finding abstractions that lead to efficient distance estimation functions. It was pioneered by Edelkamp et al. [4, 5] who proposed to base the distance estimation on the graph distance (see also the example in the previous section). This is a rather coarse abstraction that leads to distance estimation functions that are easy to compute on the one hand, but that are not very informative on the other hand. Kupferschmid et al. [12] have introduced two abstract distance functions based on the *monotonicity abstraction*. This abstraction technique is an adaptation of the *ignoring-delete-lists* principle that has originally been introduced in the area of AI Planning [1]. The basic idea is that every state variable, once it obtained a value, keeps that value forever. Therefore, the value of a variable is no longer an element, but a subset of its domain. The distance estimation values are then computed by iteratively applying transitions under this abstraction until an error state is reached, and then returning the abstract error length as estimation.

Furthermore, Dräger et al. [3] iteratively "merge" a pair of automata, i. e., compute their product and then merge locations until there are at most n locations left, where n is an input parameter. The distance estimation function is read off the overall merged automaton. Moreover, several distance estimation functions based on *pattern databases* have been proposed [9, 13, 17, 18]. A pattern database heuristic function abstracts a problem by ignoring some of the relevant symbols, e. g., some of the state variables. The state space of the abstracted problem is built completely as a pre-process to search, and is used as a look-up table for the heuristic values during search.

The problem of evaluating state transitions has been studied mostly in the area of AI Planning. In this context, an approach to avoid *useless actions* has been proposed which has led to a significantly improved search behavior on a wide range of planning instances [20]. In Section 4, we adapt this technique to the context of directed model checking of concurrent systems with interleaving and binary synchronization. Complementary to useless actions, Hoffmann and Nebel [8] and Helmert [7] proposed what they call *helpful actions* and *preferred operators*, respectively. These methods are used

to select a set of promising successors to a search state. The helpfulness of a transition is determined during the computation of the distance estimation values. These values are obtained by solving an abstract problem. Roughly speaking, a transition is considered as helpful if it is contained in that abstract solution. However, this approach is specific to the applied distance function.

3 Preliminaries

In this section, we give the basic notation as well as a formal definition of the considered models. Section 3.2 introduces directed model checking, A* and greedy search.

3.1 Notation

In our setting, an automaton is a tuple $A = (S, s^0, T, \Sigma)$, where S is a finite set of states, $s^0 \in S$ is the initial state, $T \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is a set of labeled transitions, Σ a finite set of synchronization labels, and $\tau \notin \Sigma$ denotes a special internal label. A transition $(s, \alpha, s') \in T$ is also denoted by $s \xrightarrow{\alpha} s'$.

Let N be the set $\{1, \ldots, n\}$. For n automata $A_i = (S_i, s_i^0, T_i, \Sigma_i)$, $i \in N$, with pairwise disjoint sets of states, the parallel composition $A_1 \| \ldots \| A_n$ is defined by the product automaton $(S^\times, (s_1^0, \ldots, s_n^0), T^\times, \Sigma_1 \cup \cdots \cup \Sigma_n)$, where $S^\times = S_1 \times \cdots \times S_n$, and the set of transitions $T^\times \subseteq S^\times \times \{\tau\} \times S^\times$ is defined as follows. There is a transition $(s_1, \ldots, s_n) \xrightarrow{\tau} (s'_1, \ldots, s'_n) \in T^\times$ iff one of the following conditions holds.

1. There exists $i \in N$ such that $s_i \xrightarrow{\tau} s'_i \in T_i$, and $s_k = s'_k$ for all $k \in N \setminus \{i\}$.
2. There exist $i, j \in N$ with $i \neq j$, and there exists a label $\alpha \in (\Sigma_i \cap \Sigma_j)$ such that $s_i \xrightarrow{\alpha} s'_i \in T_i$ and $s_j \xrightarrow{\alpha} s'_j \in T_j$, and $s_k = s'_k$ for all $k \in N \setminus \{i, j\}$.

A system $\mathcal{S}$ of n automata $A_1, \ldots, A_n$ is the parallel composition $A_1 \| \ldots \| A_n$. Note that in a parallel system only τ transitions occur; two synchronized transitions in automata A_i and A_j also correspond to a τ transition in $\mathcal{S}$. We address the falsification of invariants; in CTL, these properties take the form $A\,G\,\varphi$. In this paper, φ is a formula of the form $\bigwedge_i \neg s_i$, where $s_i \in S_1 \times \cdots \times S_n$ are *error* states. We call a tuple $\mathcal{T} = \langle \mathcal{S}, \varphi \rangle$ a model checking task. A *trace* $\pi = s_0, t_1, s_1, \ldots, t_n, s_n$ is an alternating sequence of states and transitions where t_i is an outgoing transition of s_{i-1}. We call a trace that leads to a state that satisfies $\neg\varphi$ an *error trace*. The length of a trace $|\pi|$ is defined as the number of transitions in π, i.e., $|\pi| = n$.

3.2 Directed Model Checking

Directed model checking describes the task of finding error states where the state space traversal is guided ("directed") by a distance estimation function $d^\#$. This function is computed fully automatically based on the declarative description of the system and an abstraction principle (e.g., the monotonicity abstraction [12]). In a nutshell, $d^\#$ is a function that maps states to integers, reflecting an estimate of the shortest error distance. Typically, this estimate is the length of a corresponding abstract error trace. States are evaluated with $d^\#$, states with lower values are preferred. Note that abstract distance functions influence the *order* in which the states are explored, thereby completeness

is *not* affected. On the one hand, it is desirable to have distance functions that are as informative as possible. On the other hand, the computation must not be too expensive.

Figure 2 shows a basic directed model checking algorithm. Given a model checking task $\langle \mathcal{S}, \varphi \rangle$ and an abstract distance function $d^{\#}$, the algorithm returns False if there is a state that violates φ, otherwise it returns True. The initial state of $\mathcal{S}$ is s^0. The algorithm maintains a priority queue open which contains visited but not yet explored states. When open.getMinimum is called open returns a minimum element, i.e., one of its elements with minimal priority value. States that have been expanded are stored in close. Every state encountered during search is first checked if it is an error state. If this is not the case, its successors are computed. Every successor that has not been visited before is inserted into open according to its priority value. The evaluate function depends on the applied version of directed model checking, i.e., if applied with A* or greedy search. For A*, evaluate$(s, d^{\#})$ returns $d^{\#}(s) + c(s)$, where $c(s)$ is the length of the path on which s was reached for the first time. For greedy search, it simply evaluates to $d^{\#}(s)$. When every successor has been computed and prioritized, the process continues with the next state from open with lowest priority value. Every state stores information about how it has been reached, i.e., its immediate predecessor state and transition. Therefore, if an error state s is finally reached, the corresponding error trace is generated by back tracing from s.

```
1 function verify(S, φ, d#):
2     open = empty priority queue, closed = ∅
3     priority = evaluate(s⁰, d#)
4     open.insert(s⁰, priority)
5     while open is not empty:
6         s = open.getMinimum()
7         if s violates φ:
8             return False
9         if s ∉ closed:
10            closed = closed ∪ {s}
11            for each outgoing transition t of s:
12                s′ = successor(s, t)
13                if s′ ∉ closed ∪ open:
14                    priority = evaluate(s′, d#)
15                    open.insert(s′, priority)
16    return True
```

Fig. 2. A basic directed model checking algorithm

For distance estimation functions that are *admissible*, i.e., that never overestimate the real error distance, A* is guaranteed to find shortest possible error traces [16].

4   Transition-Based Directed Model Checking

Until now, directed model checking algorithms have roughly followed the scheme as outlined in the last section by evaluating *states*, thereby suffering from the fact that A* is often not practical and the error traces of greedy search are often of poor quality.

In this section, we propose an extension based on transition evaluation. We will first define the theoretical concept of useless transitions and then its practical counterpart, the relatively useless transition. This notion can be directly used to combine A* and greedy search to a new *transition-based* directed model checking algorithm.

4.1 Useless and Relatively Useless Transitions

In this section, we give the definition of useless transitions. We will first give an exact notion that captures precisely our intuition on the one hand, but is computationally hard on the other hand. Therefore, we will investigate ways to approximate this definition, leading to the concept of relatively useless transitions.

Intuitively, a transition is useless if it is not needed to reach the nearest error state on a shortest path. This is formally stated in the next definition.

Definition 1 (Useless Transition). *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task, where $\mathcal{S} = (S, s^0, T, \Sigma)$. A transition $t \in T$ leading from a state s to state s' is* useless *in s iff no shortest trace from s to a nearest error state starts with this transition.*

We use $d(s)$ to denote the distance of a state s to a nearest error state. More precisely, $d(s) = n$ if there is a trace π from s to an error state with $|\pi| = n$ and there is no trace π' from s to an error state with $|\pi'| < n$. When we want to stress that d is a function also on the system $\mathcal{S}$, we will write $d(\mathcal{S}, s)$.

By Definition 1, a transition t is useless in a state s if and only if the real error distance d does not decrease by one, i. e., a transition from s to s' is useless iff $d(s) \leq d(s')$. To see this, recall that $d(s) \leq d(s') + 1$ for every transition. If a shortest error trace starts from s with t, then $d(s) = d(s') + 1$. Otherwise the error distance does not decrease, i. e., $d(s) < d(s') + 1$. Since the distance values are all integers, this is equivalent to $d(s) \leq d(s')$. We will use the inequality $d(s) \leq d(s')$ in connection with the idea of *removing* a useless transition. Therefore, we will define the notion of *reduced systems*. To do this, we first need some more terminology. For a system $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$, say $\mathcal{S} = (S, s^0, T, \Sigma)$, we define a function $\mu_{\mathcal{S}}$ that maps transitions from the system $\mathcal{S}$ to the corresponding transitions in $A_i = (S_i, s_i^0, T_i, \Sigma_i)$. This function $\mu_{\mathcal{S}} : T \to 2^{T_1 \cup \cdots \cup T_n}$ is defined as follows. When we write $s_i \xrightarrow{c} s_i'$ for a transition t, we assume that t is contained in automaton i, i. e., $s_i \xrightarrow{c} s_i' \in T_i$.

$$\mu_{\mathcal{S}}((s_1, \ldots, s_n) \xrightarrow{\tau} (s_1', \ldots, s_n'))$$
$$= \begin{cases} \{s_i \xrightarrow{\tau} s_i'\} & \exists i \in \{1, \ldots, n\} \\ \{s_i \xrightarrow{\alpha} s_i', s_j \xrightarrow{\alpha} s_j'\} & \exists i, j \in \{1, \ldots, n\}, i \neq j, \exists \alpha \in (\Sigma_i \cap \Sigma_j) \end{cases}$$

Based on this definition, we now define *reduced systems*.

Definition 2 (Reduced system). *Let $\mathcal{S} = A_1 \parallel \ldots \parallel A_n$ be a system, say $\mathcal{S} = (S, s^0, T, \Sigma)$, with $A_i = (S_i, s_i^0, T_i, \Sigma_i)$ for $i \in \{1, \ldots, n\}$. Let $t \in T$ be a transition. The* reduced system *with respect to t is defined as $\mathcal{S}_t = A_1' \parallel \ldots \parallel A_n'$, where $A_i' = (S_i, s_i^0, T_i \setminus \mu_{\mathcal{S}}(t), \Sigma_i)$.*

Note that, according to the definition of $\mu_{\mathcal{S}}$, at most two automata A_i are affected by reducing the system (one in the case of interleaving, two in the case of binary synchronization). Roughly speaking, a transition t of the system $\mathcal{S}$ corresponds to one or two

transitions of one or two automata of $\mathcal{S}$. The reduced system $\mathcal{S}_t$ is obtained by removing these transitions from the corresponding automata. Note that removing *one* transition from an automaton A removes *several* transitions from the system $\mathcal{S}$.

Based on the definition of reduced systems, we will give a proposition that leads to a testing criterion for useless transitions.

Proposition 1. *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task with $\mathcal{S} = (S, s^0, T, \Sigma)$, $s, s' \in S$, $t \in T$ leading from s to s'. If $d(\mathcal{S}_t, s) \leq d(\mathcal{S}, s')$, then t is useless in s.*

Proof. If $d(\mathcal{S}_t, s) \leq d(\mathcal{S}, s')$, then $d(\mathcal{S}, s) \leq d(\mathcal{S}, s')$ because $d(\mathcal{S}, s) \leq d(\mathcal{S}_t, s)$ (the error distance cannot decrease in reduced systems). As $d(s) \leq d(s')$ iff t is useless in s, the claim follows directly.

This characterization can be interpreted as follows. A transition t is useless in s if the error state is still reachable from s on the same shortest trace when the corresponding transitions to t are *removed* from the system. However, it is not practical as computing exact distances is PSPACE-hard. A direct way to approximate this test is to use the given distance estimation function $d^{\#}$ instead of d. This is rational because $d^{\#}$ is designed for exactly the reason of approximating d. When we want to stress that $d^{\#}$ is a function also on the system S, we will write $d^{\#}(\mathcal{S}, s)$.

Definition 3 (Relatively Useless Transition). *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task with $\mathcal{S} = (S, s^0, T, \Sigma)$, $s, s' \in S$, $t \in T$ leading from s to s'. Let $d^{\#}$ be a distance estimation function. Then t is* relatively useless *for $d^{\#}$ in s if $d^{\#}(\mathcal{S}_t, s) \leq d^{\#}(\mathcal{S}, s')$.*

Note that this is exactly the testing criterion from Proposition 1 where d has been replaced by $d^{\#}$. Obviously, the quality of this approximation strongly depends on $d^{\#}$'s precision. A very uninformed function, e. g. a function that constantly returns zero, recognizes every transition as relatively useless. However, the more sophisticated the distance estimation, the more precise is the approximation. We will come back to this point in the next section. Intuitively, taking a relatively useless transition t does not seem to guide the state space traversal towards an error state as the *stricter* distance estimate in $\mathcal{S}_t$ does not increase.

One would expect that transitions should not be relatively useless if they lead to states nearer to an error state. Indeed, under the rational assumption that distance functions $d^{\#}$ never decrease their estimate in reduced systems, i. e., $d^{\#}(\mathcal{S}, s) \leq d^{\#}(\mathcal{S}_t, s)$ for all systems $\mathcal{S}$, transitions t and states s, transitions leading to better estimates are *never* relatively useless in any system $\mathcal{S}$.

Proposition 2. *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task with $\mathcal{S} = (S, s^0, T, \Sigma)$. Let $d^{\#}$ be a distance estimation function such that $d^{\#}(\mathcal{S}, s) \leq d^{\#}(\mathcal{S}_t, s)$ for all $s \in S$ and $t \in T$. Let $s, s' \in S$ be states and $t \in T$ be a transition that leads from s to s'. If $d^{\#}(s') < d^{\#}(s)$, then t is* not *relatively useless for $d^{\#}$ in s.*

Proof. Assume that t is relatively useless, i. e., $d^{\#}(\mathcal{S}_t, s) \leq d^{\#}(\mathcal{S}, s')$. As $d^{\#}(\mathcal{S}, s) \leq d^{\#}(\mathcal{S}_t, s)$, we have $d^{\#}(\mathcal{S}, s) \leq d^{\#}(\mathcal{S}, s')$, showing that the distance estimate does not decrease when the relatively useless transition t is applied.

4.2   Directed Model Checking with Relatively Useless Transitions

In this section, we put the pieces together. So far, we have presented a notion of useless transitions to identify transitions that should be less preferred during the state space traversal. A direct way to integrate this information is to "penalize" states that result from applying such a transition. This is rational because avoiding transitions that are not likely to appear in shortest error traces is likely to improve the detection of (short) error traces. States that are reached by applying such a useless transition should be less preferred when traversing the state space.

As argued in the introduction and Section 3.2, there are two choices to be made when the directed model checking approach is applied, namely choosing the underlying abstraction for the distance estimation functions, and choosing the algorithm that is essentially determined by the *evaluate* function that computes the priority values for the states. Here, we assume that a distance estimation function $d^\#$ is already given, and $d^\#$ is additionally used to determine relatively useless transitions. For the second point, we give a simple extension of the *evaluate* function in Fig. 3. Recall that s and t (lines 2 and 3) are stored in the successor state and can be accessed easily. As outlined above, states that result from applying a relatively useless transition are "penalized". As penalty value for s, we chose $c(s)$, the length of the trace on which s was reached for the first time. This leads to a combination of A^* and greedy search as discussed in more detail below.

```
1 function evaluate(s', d#):
2     s = predecessor of s'
3     t = transition from s to s'
4     if t is relatively useless for d# in s:
5         priority = d#(s') + c(s')
6     else:
7         priority = d#(s')
8     return priority
```

Fig. 3. Evaluation function based on relatively useless transitions

Overall, this algorithm is an amalgam of the algorithms A^* and greedy search based on transition evaluation. Its behavior depends on the accuracy of the underlying distance estimation function $d^\#$. As mentioned earlier, the more accurate $d^\#$, the more transitions are classified correctly, and therefore, the more it tends towards greedy search. At the extreme ends of the spectrum, it becomes greedy search (for the perfect distance function that classifies every transition correctly) and breadth first search, respectively, which is a degenerated version of A^* for the distance function that constantly returns zero. From this perspective, our algorithm can be considered as a combination of greedy search and A^*.

4.3   Discussion

Although it is technically possible to apply our algorithm to every model checking task, there are distance functions that are probably best suited for this concept. Let us

have a look at this class of functions. Roughly speaking, distance estimation functions can be divided into two classes, namely those that compute the values on-the-fly by solving an abstract problem in every search state, and those that do it in a preprocessing step, typically by computing a lookup table (e. g., a pattern database). The concept of useless transitions seems to be best suited for distance functions that are computed on-the-fly because the time overhead to compute this information is comparatively low. Contrarily, distance functions from the second class are less suited because for every modified system, an additional pattern database has to be computed (recall that for the computation of the useless-values, the system is modified and the distance value is recomputed on this modified system). However, as we will see, for distance functions computed on-the-fly, the overall performance can often be significantly improved.

The performance of our approach strongly depends on the quality of $d^{\#}$ that is used to guide the search and to determine useless transitions. The higher the precision of $d^{\#}$, the more transitions are evaluated correctly, and hence, the better the overall performance as many unnecessary states need not be considered. In small examples, distance functions like the graph distance could already lead to improvements. Pointing to our motivating example in the introduction, we recognize that all transitions corresponding to edges from down to up are relatively useless for the graph distance heuristic, whereas all other transitions are not. In this example, applying our algorithm leads to a shortest possible error trace with dramatically smaller explored state space than with greedy search or A^{*}. For more complex examples, more sophisticated distance functions are needed to benefit from our approach, as we will empirically show in the next section.

5   Evaluation

We have implemented our algorithm in the model checker MCTA [14] as part of a tool development effort within the AVACS project[1]. The tool and its source code are freely available at `http://mcta.informatik.uni-freiburg.de/`. We compare our search method with A^{*} and greedy search. In addition to the automaton model as considered in this paper, many of them feature integer and clock variables and represent timed automata. Transitions can additionally be guarded by integer and clock constraints. Moreover, a transition can change the value of integer variables and reset clock variables. Note that the concept of useless transitions is general and can be adapted to that class of automata in a straightforward way. To get a conservative approximation of useless transitions, we have implemented our concept in a stronger way than described in the last section. When the reduced system $\mathcal{S}_t$ is computed for a system $\mathcal{S} = A_1 || \ldots || A_n$ and a transition t in $\mathcal{S}$, we *additionally* remove transitions in the automata A_i that read variables that are set by some $t' \in \mu_{\mathcal{S}}(t)$, and transitions that lead to the same state as some $t' \in \mu_{\mathcal{S}}(t)$.

5.1   The Distance Estimation Functions

We evaluated our algorithm for a number of distance estimation functions. We give detailed results for the distance functions h^L and h^U introduced by Kupferschmid et al.

---

[12] and for the distance function based on the maximum graph distance h^{gd} introduced by Edelkamp et al. [4, 5]. As outlined in Section 2, h^L and h^U are based on the monotonicity abstraction principle, where a state variable can have multiple values simultaneously. h^L performs a fixpoint iteration under this abstraction starting in the current state until an error state is reached, and returns the number of iterations as distance estimate. Based on this fixpoint iteration, h^U additionally extracts an abstract error trace starting from the abstract error state, and returns the number of abstract transitions as the estimate. Observe that computing h^U is more expensive than h^L. As we will see in Section 5.3, this pays off in better search behavior. The maximum graph distance function h^{gd} uses the graph distance as indicated by its name.

5.2 The Benchmark Set

Our benchmarks stem from the AVACS benchmark suite.

Industrial benchmarks. The M and N examples come from a case study that models a real-time protocol to ensure mutual exclusion of a state in a distributed system via asynchronous communication. The protocol is described in full detail in [2]. The C examples stem from a case study from an industrial project partner of the UniForM-project [10] where the problem is to design a distributed real-time controller for a segment of tracks where trams share a piece of track. For the evaluation of our approach we chose the property that both directions are never given simultaneous permission to enter the shared segment. In both case studies, a subtle error has been inserted by manipulating a delay so that the asynchronous communication between these automata is faulty.

Academic benchmarks. The F^A and F^B examples are flawed versions of the *Fischer protocol* for mutual exclusion (cf. [15]). The variants differ in the way they encode the error condition. As a second set of benchmarks, we use *arbiter trees* to establish mutual exclusion between 2^k client processes [19]. The benchmarks A_2–A_6 contain arbiter trees of height 2–6, with an exponentially growing number of processes.

5.3 Experimental Results

The reported experimental results were obtained on a 2.66 GHz Intel Xeon computer with memory out at 4 GB and a Linux operating system. We compare our new state space traversal technique, denoted *UT*, with A* and greedy search (G) in three different configurations. In the first configuration, h^L is used as the abstract distance function, the second uses h^U and the third configuration uses h^{gd}.

Table 1 shows the results of the first configuration. Here, the number of explored states significantly decreases compared to A* and we are able to solve much larger problems. Compared to greedy search, the length of the found error traces are significantly shorter. Moreover, due to better search guidance, we additionally often get significant improvements in terms of the number of explored states and traversal time.

The results for the second configuration are depicted in Table 2. Note that h^U is more informative than h^L, and search behavior therefore is mostly better (in particular, the Fischer protocol examples are trivial for h^U). This fact directly influences the performance when applied with our algorithm: With *UT* and h^U, we obtain even better

Table 1. Experimental results for h^L with A*, greedy search (G), and our combined approach (UT). Abbreviations: #a: number of parallel automata, #v: number of variables, *memory*: peak memory used in MB, $y\,\mathrm{e}{+}x\colon y \cdot 10^x$, dashes indicate out of memory ($> 4\,\mathrm{GB}$).

Inst.	#a	#v	explored states			runtime in s			memory			trace length		
			A*	G	UT	A*	G	UT	A*	G	UT	A*	G	UT
C_1	5	15	22501	1928	1658	0.16	0.04	0.06	9	8	8	54	100	91
C_2	6	17	66791	4566	1333	0.48	0.09	0.08	14	8	8	54	132	91
C_3	6	18	76777	6002	1153	0.58	0.11	0.06	15	9	8	54	128	91
C_4	7	20	726516	81131	1001	5.34	1.00	0.10	70	19	8	55	344	121
C_5	8	22	6.00e+6	430494	833	44.64	5.32	0.12	484	63	8	56	1057	114
C_6	9	24	–	4.56e+6	833	–	48.00	0.17	–	521	9	–	3217	114
C_7	10	26	–	–	829	–	–	0.22	–	–	9	–	–	114
C_8	10	27	–	1.19e+7	816	–	110.81	0.18	–	1158	9	–	5644	95
C_9	10	28	–	2.77e+7	13423	–	252.66	2.27	–	2534	22	–	5803	90
M_1	3	15	34680	4581	4256	0.17	0.02	0.02	9	8	8	47	457	97
M_2	4	17	135073	15832	8186	0.75	0.08	0.06	15	10	9	50	1124	146
M_3	4	17	155164	7655	10650	0.88	0.04	0.07	15	9	10	50	748	91
M_4	5	19	584221	71033	22412	4.27	0.44	0.19	38	19	15	53	3381	136
N_1	3	18	80541	50869	5689	0.97	1.26	0.06	17	45	9	49	26053	108
N_2	4	20	332486	30476	22763	5.00	0.31	0.24	37	19	14	52	1679	259
N_3	4	20	406908	11576	35468	6.66	0.12	0.42	38	12	15	52	799	204
N_4	5	22	1.59e+6	100336	142946	33.78	1.14	1.86	117	40	39	55	2455	792
F_5^A	6	6	71	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_{10}^A	11	11	511	9	9	0.00	0.00	0.00	8	7	7	8	8	8
F_{15}^A	16	16	1701	9	9	0.05	0.00	0.00	16	7	7	8	8	8
F_5^B	5	6	54	179	7	0.00	0.00	0.00	7	7	7	6	12	6
F_{10}^B	10	11	429	86378	7	0.01	1.29	0.00	8	109	7	6	22	6
F_{15}^B	15	16	1504	–	7	0.04	–	0.00	17	–	7	6	–	6
A_2	8	0	73	36	15	0.00	0.00	0.00	7	7	7	12	21	12
A_3	16	0	5168	206	32	0.08	0.01	0.01	8	7	7	17	24	17
A_4	32	0	4.44e+6	76811	95	95.95	7.53	0.06	1060	65	9	22	42	22
A_5	64	0	–	263346	34	–	50.83	0.11	–	325	13	–	112	27
A_6	128	0	–	–	39	–	–	0.49	–	–	30	–	–	32

results than with UT and h^L. Note that h^U is not admissible, which means that there is no guarantee to obtain a shortest possible error trace in this setting in theory. However, in practice, we obtained shortest possible traces in our examples with A*. The trace length of UT is still mostly shorter than with greedy search. In particular, note that for both h^L and h^U configurations without UT, the large C examples C_7–C_9 could only be solved with an error trace of very poor quality compared to UT. Moreover, the largest arbiter example A_6 could not be solved at all without UT within 4 GB of memory.

Table 3 gives the results for the third configuration (h^{gd}). Here, we observe that the results with UT are less significant than with the first two configurations. This is because, having a closer look at the distance estimation values in many of the instances, the estimation values are often constant. This is due to the very coarse abstraction (i. e., the graph distance) used by h^{gd}. Therefore, too many transitions are relatively useless for this distance function, causing the search process to degenerate towards A*.

Table 2. Experimental results for h^U. Abbreviations as in Table 1.

Inst.	#a	#v	explored states			runtime in s			memory			trace length		
			A*	G	UT	A*	G	UT	A*	G	UT	A*	G	UT
C_1	5	15	12480	715	277	0.22	0.02	0.02	9	7	7	54	73	59
C_2	6	17	35047	1612	242	0.56	0.05	0.03	12	7	7	54	99	59
C_3	6	18	39755	734	228	0.68	0.03	0.03	12	7	7	54	86	59
C_4	7	20	359376	9120	566	5.46	0.15	0.12	50	9	8	55	139	55
C_5	8	22	2.88e+6	83911	190	42.01	1.08	0.06	325	18	8	56	300	56
C_6	9	24	2.89e+7	718015	190	374.72	6.39	0.08	3122	79	8	56	864	56
C_7	10	26	–	2.55e+6	184	–	21.74	0.10	–	236	8	–	2412	56
C_8	10	27	–	1.11e+7	570	–	145.24	0.28	–	1237	9	–	3733	94
C_9	10	28	–	–	1225	–	–	0.67	–	–	10	–	–	153
M_1	3	15	33999	7668	4366	0.16	0.04	0.03	9	8	8	47	71	73
M_2	4	17	124237	18847	2036	0.71	0.11	0.02	14	10	8	50	119	81
M_3	4	17	157173	19597	12829	0.93	0.11	0.11	16	10	11	50	124	89
M_4	5	19	562527	46170	9873	4.30	0.28	0.11	40	16	12	53	160	97
N_1	3	18	78798	9117	5191	0.96	0.08	0.05	17	9	9	49	99	80
N_2	4	20	279853	23462	3260	4.17	0.24	0.04	35	15	9	52	154	136
N_3	4	20	378963	43767	19271	6.16	0.47	0.22	39	20	14	52	147	149
N_4	5	22	1.32e+6	152163	15102	26.91	1.97	0.20	110	54	18	55	314	377
F_5^A	6	6	9	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_{10}^A	11	11	9	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_{15}^A	16	16	9	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_5^B	5	6	7	7	7	0.00	0.00	0.00	7	7	7	6	6	6
F_{10}^B	10	11	7	7	7	0.00	0.00	0.00	7	7	7	6	6	6
F_{15}^B	15	16	7	7	7	0.00	0.00	0.00	7	7	7	6	6	6
A_2	8	0	20	25	20	0.00	0.01	0.00	7	7	7	12	21	18
A_3	16	0	25	82	27	0.00	0.01	0.01	7	7	7	17	18	17
A_4	32	0	213	39	34	0.06	0.02	0.05	9	8	9	22	28	22
A_5	64	0	187148	4027	42	42.68	1.22	0.23	414	17	13	27	47	27
A_6	128	0	–	–	50	–	–	1.10	–	–	31	–	–	32

However, *UT* mostly still explores less states than A*, thereby producing *significant* shorter error traces than greedy search.

Overall, the concept of useless transitions has shown its potential in an impressive way. The results show a significant improvement of the error traces in comparison to greedy search as well as a significant reduction of the explored state space compared to A*. On many problems, the size of the explored state space is even lower than with greedy search. Our experimental evaluation has shown this effect on a large number of benchmarks, ranging from academic to industrial examples with instances of different difficulties, ranging from very easy to very hard. Some problems represent timed systems. We have seen that the overall performance of *UT* depends on the precision of the underlying distance estimation function. With *UT*, a sophisticated distance function like h^L already often leads to significant better guidance of the state space traversal than with greedy search and A*. More informative distance functions (like h^U) also lead to better search guidance when applied with *UT*, and hence, the number of explored states further decreases. With less informative distance functions (like h^{gd}), the impact of *UT* decreases and the whole search process degenerates towards A*.

Table 3. Experimental results for h^{gd}. Abbreviations as in Table 1.

Inst.	#a	#v	explored states			runtime in s			memory			trace length		
			A^*	G	UT	A^*	G	UT	A^*	G	UT	A^*	G	UT
C_1	5	15	56496	18796	32583	0.12	0.06	0.12	11	9	10	54	1167	61
C_2	6	17	185109	66389	107175	0.49	0.22	0.42	20	14	17	54	1847	69
C_3	6	18	240090	94536	133529	0.68	0.33	0.55	24	17	20	54	2153	68
C_4	7	20	2.45e+6	1.11e+6	1.27e+6	8.00	3.80	5.84	160	100	124	55	6805	71
C_5	8	22	2.28e+7	1.27e+7	1.07e+7	82.79	43.50	56.38	1319	877	976	56	35067	67
C_6	9	24	–	–	–	–	–	–	–	–	–	–	–	–
M_1	3	15	44611	12277	19333	0.21	0.07	0.10	9	9	8	47	2779	95
M_2	4	17	176429	43784	67184	0.96	0.28	0.33	16	14	11	50	11739	105
M_3	4	17	188472	54742	84020	1.05	0.37	0.46	15	15	12	50	12701	113
M_4	5	19	706127	202924	319485	5.02	1.69	1.98	41	43	26	53	51402	218
N_1	3	18	94908	15732	29276	1.10	0.18	0.28	17	11	11	49	3565	113
N_2	4	20	391813	102909	149431	5.49	1.42	1.71	36	27	22	52	18180	130
N_3	4	20	428812	131202	166041	6.34	2.04	1.94	35	30	21	52	20021	160
N_4	5	22	1.76e+6	551091	734171	34.13	11.73	10.89	111	115	74	55	90467	147
F_5^A	6	6	658	271	658	0.00	0.00	0.00	7	7	7	8	218	8
F_{10}^A	11	11	13623	271	13623	0.12	0.00	0.13	24	8	24	8	218	8
F_{15}^A	16	16	109773	271	109773	1.61	0.00	1.78	309	9	309	8	218	8
F_5^B	5	6	78	496	9	0.00	0.00	0.00	7	7	7	6	79	6
F_{10}^B	10	11	523	6.73e+6	9	0.00	94.81	0.00	8	3254	7	6	27107	6
F_{15}^B	15	16	1718	–	9	0.03	–	0.00	17	–	7	6	–	6
A_2	8	0	359	27	334	0.00	0.01	0.00	7	7	7	12	22	12
A_3	16	0	61633	344	49652	0.13	0.01	0.18	13	7	14	17	169	17
A_4	32	0	–	38209	–	–	0.30	–	–	18	–	–	867	–
A_5	64	0	–	–	–	–	–	–	–	–	–	–	–	–

6   Conclusion

We have introduced the concept of useless transitions to directed model checking as an adaptation of the useless actions approach that has successfully been proposed in the area of AI Planning. Based on useless transitions, we have proposed a hybrid algorithm between A^* and greedy search that seems to identify the sweet spot of the trade-off between scalability and short computed error traces. We have implemented this algorithm and evaluated it empirically on a number of benchmarks for a number of distance estimation functions. Our empirical evaluation shows a substantial performance gain in terms of explored states when compared with A^*, and a significant solution quality improvement when compared with greedy search. Due to better guidance abilities, we often even explore less states than greedy search, being able to solve much larger problems than A^* and greedy search.

As outlined in the discussion section, our approach seems to be currently best suited for distance estimation functions that are computed on-the-fly, and less suited for distance functions based on pattern databases. This is because the time overhead seems to be too large when adapting it for such functions in a straight forward way. To investigate how to adapt our concept *efficiently* to distance functions based on pattern databases will be an important topic for future research. Furthermore, it will be interesting to refine our

concept to more than two degrees of uselessness. We expect that algorithms exploiting that knowledge further improve the state space traversal.

Acknowledgments

References

1. Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence 129(1–2), 5–33 (2001)
2. Dierks, H.: Comparing model-checking and logical reasoning for real-time systems. Formal Aspects of Computing 16(2), 104–120 (2004)
3. Dräger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 19–34. Springer, Heidelberg (2006)
4. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. International Journal on Software Tools for Technology Transfer 5(2), 247–267 (2004)
5. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
6. Groce, A., Visser, W.: Heuristics for model checking Java programs. International Journal on Software Tools for Technology Transfer 6(4), 260–276 (2004)
7. Helmert, M.: The Fast Downward planning system. Journal of Artificial Intelligence Research 26, 191–246 (2006)
8. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302 (2001)
9. Hoffmann, J., Smaus, J.-G., Rybalchenko, A., Kupferschmid, S., Podelski, A.: Using predicate abstraction to generate heuristic functions in Uppaal. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS, vol. 4428, pp. 51–66. Springer, Heidelberg (2007)
10. Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Baer, A.: The $UniForM\,workbench$, a universal development environment for formal methods. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, pp. 1186–1205. Springer, Heidelberg (1999)
11. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: UPPAAL/DMC – abstraction-based heuristics for directed model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 679–682. Springer, Heidelberg (2007)
12. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006)
13. Kupferschmid, S., Hoffmann, J., Larsen, K.G.: Fast directed model checking via russian doll abstraction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 203–217. Springer, Heidelberg (2008)

14. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster than UPPAAL? In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 552–555. Springer, Heidelberg (2008)
15. Lamport, L.: A fast mutual exclusion algorithm. ACM Transactions on Computer Systems 5(1), 1–11 (1987)
16. Pearl, J.: Heuristics: Intelligent search strategies for computer problem solving. Addison-Wesley, Reading (1984)
17. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)
18. Qian, K., Nymeyer, A., Susanto, S.: Abstraction-guided model checking using symbolic IDA* and heuristic synthesis. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 275–289. Springer, Heidelberg (2005)
19. Seitz, C.L.: Ideas about arbiters. Lambda 1, 10–14 (1980)
20. Wehrle, M., Kupferschmid, S., Podelski, A.: Useless actions are useful. In: Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008), pp. 388–395. AAAI Press, Menlo Park (2008)

Memoised Garbage Collection
for Software Model Checking

Viet Yen Nguyen[1] and Theo C. Ruys[2]

[1] RWTH Aachen University, Germany
http://moves.rwth-aachen.de/~nguyen/
[2] University of Twente, The Netherlands
http://www.cs.utwente.nl/~ruys/

Abstract. Virtual machine based software model checkers like JPF and MOONWALKER spend up to half of their verification time on garbage collection. This is no surprise as after nearly each transition the heap has to be cleaned from garbage. To improve this, this paper presents the Memoised Garbage Collection (MGC) algorithm, which exploits the (typical) locality of transitions to incrementally perform garbage collection. MGC tracks the depths of objects efficiently and only purges objects whose depths have become infinite, hence unreachable. MGC was experimentally evaluated via an implementation in our model checker MOONWALKER and benchmarks using the parallel Java Grande Forum benchmark suite. By using MGC, a performance increase up to 78% was measured over the traditional Mark&Sweep implementation.

1 Introduction

Within the software development cycle, model checkers are often used to validate the initial design of a system before actually implementing it. The process of model checking usually consists of three parts: modelling, specification and verification. During the modelling phase, an abstraction is made from the design under verification. This abstraction – the model – is then verified against the specification. This traditional approach has its disadvantages. For, (i) creating an abstraction at the right level is considered difficult, (ii) the abstraction is crafted manually and prone to human error, (iii) the model and its semantics are bounded to the expressiveness of the modelling language, which generally tend to be rigourously formalised (e.g., process algebra's, state machines) [4] and (iv) after validation, the model still has to be transformed to an implementation. As fully automated code generators do not exist (yet), parts of this refinement step have to be done manually.

Software model checking overcomes these labor intensive problems by verifying the implemented system directly instead of the abstract model. This approach has been pioneered by Klaus Havelund [9] in the first version of the Java PathFinder (JPF). This initial version of JPF comprised a Java to Promela translator that enabled the verification of Java programs using the model checker SPIN [10]. This experiment highlighted many challenges associated with the
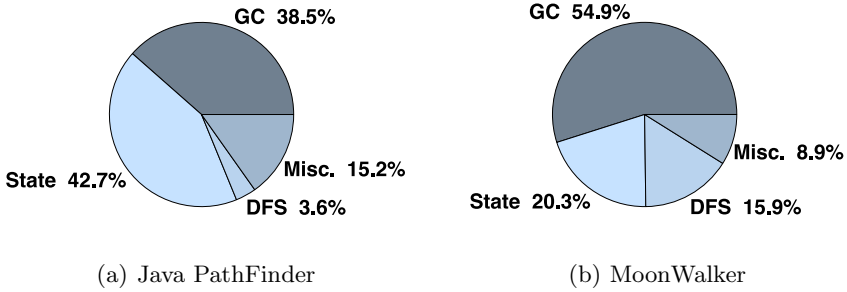
(a) Java PathFinder   (b) MoonWalker

Fig. 1. Profiler data from Raytracer 3-1 benchmark [22] (see Section 4.2) describing the stakes of garbage collection (GC), state storage (State), exploration (DFS) and all remaining functionality (Misc.)

software model checking approach. The foremost problem was overcoming the semantic gap between Java and Promela. This was solved by introducing the bytecode interpretation approach in the second iteration of JPF [23,25]. Ever since, more techniques have been developed to make software model checking more effective [15]. Within the context of this paper, the emergence of thread and heap symmetry reduction techniques [11,12,17] are most important.

These developments gave rise to other software model checkers, like XRT [8], BOGOR [20], BANDERA [4] and MOONWALKER [2,21,26]. The latter is a software model checker developed at the University of Twente. It verifies CIL assemblies – better known as Microsoft .NET programs – for assertion violations and deadlocks. It is written in C#, runs on Windows, Linux and MacOS X and its main purpose it so serve as a testbed for experimenting with novel model checking techniques. The architecture of the initial version [1] was heavily inspired by JPF's. In terms of performance, MOONWALKER is comparable with JPF. For the second iteration of MOONWALKER, we developed new techniques for speeding up verification [18]. In this paper, we present one of our contributions.

Using a profiler, we observed that the software model checkers typically spend around half of the time on garbage collection (see Figure 1 for an illustrating example). This does not come as a surprise as after most transitions the heap has to be cleaned from garbage. To lower the stake of garbage collection and therefore reduce the time needed for verification, we developed a new algorithm called the Memoised Garbage Collector (MGC). This algorithm is inspired by incremental graph updates from graph grammars and routing [19], and has a more favourable time-complexity compared to the often used Mark&Sweep (M&S) algorithm [16]. The key idea here is that instead of calculating reachability of a vertex (like M&S does), *track the depths of the objects efficiently and purge objects whose depth becomes infinite*, i.e., became unreachable.

This paper is further organised as follows. Section 2 outlines background research. This is followed by a description of MGC in section 3. Section 4 describes the benchmark setup, results and discussion. This paper ends on directions for future work (section 5) and the conclusions.

2   Background

Notation-wise, in this paper, a directed graph G with a source vertex is defined as $G = (V, v_0, E)$, where V is the set of vertices, E the set of edges, and $v_0 \in V$ is the initial, root vertex. The direct predecessors of a vertex u in a graph G is defined as the set $Pred(G, u)$. The set of direct successors of a vertex u are defined as $Succ(G, u)$.

2.1   Garbage Collection for Symmetry Reduction

Garbage collection [14] is a form of automatic memory management. It is the process of reclaiming memory allocations that will not be used in the future, thereby freeing up memory. Garbage collection is a rather expensive process, it usually requires the traversal of all memory allocations before it is decidable which allocations can be reclaimed. Within the context of software model checking, garbage collection is used for a slightly different purpose, as identified by Iosif [13].

The scenario of a typical software model checker is as follows. Consider an object-oriented language that disallows pointer arithmetic, like Java or C#. Objects used by a program are internally stored in an array. Yet, because pointer arithmetic's is disallowed, the index of an object (i.e., its address) has no semantic value. Objects can only be reached via dereferencing. When references between objects in an array are mapped, the resulting graph is a heap graph. The shape of the heap graph is of semantic value, because the references between objects are. Due to different interleavings of a program, the software model checker can reach different states such that both have the same heap graph shape, but the objects in question are permutated differently in the respective arrays. If states are matched by simply matching array-equivalence, the semantically equivalent heaps will be seen as different, thereby increasing the number of states unnecessary. Detection of semantically equivalent heaps is called *heap symmetry detection* [11,15].

To date, two variants of heap symmetry reduction are known to be effective. The technique of Iosif [11] traverses the full heap graph and creates a canonical array of objects out of it. This canonical array is stored in the hashtable. Upon state matching, the state to be matched is canonicalised and then the canonicalised arrays are matched. The technique of Lerda et al. [15] maintains a canonicalised array, instead creating one when necessary. The latter is employed by MOONWALKER. Both rely on the garbage collection algorithm to function. A heap graph traversal is needed for purging unreferencable, i.e., garbage, objects. This stems from an important observation by Iosif that garbage objects may differ between states that have different paths leading to them, but are equivalent when canonicalised [13].

In software model checking though, it is observable that changes between successive states are small. Hence, the changes to the heap graph are also small. This can be exploited by tracking these changes and have them drive the garbage collection algorithm. Time can be saved for especially large heaps.

2.2   Incremental Shortest Path

To take advantage of the small changes between successive states, we propose a garbage collection algorithm inspired by an *incremental shortest-path algorithm*. A generalised algorithm for single-source directed graphs with positive weights was devised by Ramalingam and Reps [19]. See Algorithm 1. It can be viewed as an incremental version of Dijkstra's shortest path algorithm [5].

Traditionally, depths of vertices are computed all at once using Dijkstra's algorithm, stored and used when necessary. We use $depth(u)$ to indicate the stored depth of vertex u. When the graph changes from G to G', the real depths of the vertices may change. This is however not reflected in the stored depths. Thus usually, upon a change to the graph, Dijkstra's algorithm is called to globally recompute the stored depths.

For large graphs, it is more efficient to recompute only the stored depths of vertices whose real depths have changed. To date however, there is no method to determine efficiently and precisely this set of vertices. However an over-approximation of this set can be traversed by using Ramalingam and Reps's notion of inconsistency and a top-down traversal order. The former is defined as follows:

> Given the stored depth mapping $depth$, a graph $G' = (V', v'_0, E')$ and the right-handside function $rhs(G', u) = \min_{v \in Pred(G',u)} depth(v) + 1$, a vertex $u \in V'$ is inconsistent if $rhs(G', u) \neq depth(u)$.

Inconsistent vertices are spotted cheaply by monitoring the changes to the predecessor transitions upon graph changes, as shown later in Section 3. Then, the inconsistent vertices are traversed according to their key, which is defined as the minimum of the rhs and the stored $depth$: $key(G', u) = \min(rhs(G', u), depth(u))$. The vertex u with the lowest key is processed first, see line 2 of Algorithm 1. It is the inconsistent vertex closest to the root. If there are multiple vertices with the same lowest key, one is selected non-deterministically. In case its rhs is smaller than its stored depth, we know that the changes to the graph moved u closer to the root. We can assign its rhs value to $depth$ to make it consistent (line 3-4). This could cause its successors, $Succ(G', u)$, to become inconsistent, and they will be processed when their key is the lowest. On line 5-6, we deal with the case that $rhs(G', u)$ is greater than $depth(u)$, thus it moved farther from the root. We assign its stored depth with infinity (∞). This ensures vertex u's key is purely determined by the rhs and if it is the lowest, it will be processed again.

Algorithm 1. RamalingamReps()

Data. graph $G' = (V', v'_0, E')$

1 while G' *contains inconsistent vertices* **do**
2 $u \leftarrow$ the vertex with the lowest key
3   **if** $rhs(G', u) < depth(u)$ **then**
4 $depth(u) \leftarrow rhs(G', u)$
5   **else if** $depth(u) < rhs(G', u)$ **then**
6 $depth(u) \leftarrow \infty$

The cause-and-effect behaviour of making vertices consistent and triggering its successors become inconsistent is guaranteed to reach a fixpoint because of the traversal order by the lowest key. A proof of correctness is provided in [19].

Intuitively, this algorithm determines a subgraph of vertices for which the stored depths reflect the real depths. This is ensured for consistent vertices whose stored depth is smaller or equal to the vertex with the smallest key value. Based on this subgraph, the inconsistent vertex closest to this subgraph is made consistent. This enlarges the subgraph. This is recursively done until all inconsistent vertices are traversed and the subgraph is equal to the graph. A walkthrough of this algorithm is shown in Figure 3. It outlines the steps of Ramalingam and Reps's algorithm on graph G' from Figure 2.
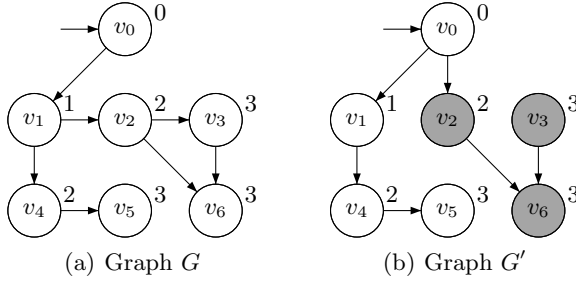
(a) Graph G (b) Graph G'

Fig. 2. Graph G was changed to graph G', but the stored depths (the labels upper-right from the vertex) were not recomputed. Because of this, vertices v_2, v_3 and v_6 are inconsistent, as indicated by the gray fill in graph G'.

The foremost application of Ramalingam and Reps's algorithm is in routing. Routers need to recalculate shortest paths to neighbouring routers when the connections change. Whereas Dijkstra's algorithm recalculates all shortest paths, this algorithm only recalculates shortest paths that have actually changed. For large networks, this algorithm reduces time. In this paper, we show how the idea behind this algorithm improves garbage collection in software model checking.

3   Memoised Garbage Collection

Ramalingam and Reps's algorithm works on graphs in general. To make it applicable for garbage collection in software model checking, we introduce additional semantics upon it.

First, a heap does not have a single root object, but multiple, namely the objects referenced from the call stacks of the program threads (see Figure 4). To make a heap graph a single-root graph, we introduce a fictive root v_0 whose successors are the objects referenced from the call stacks. Each reference counts as a distance of one. Given these semantic additions, the resulting graph can be processed by Ramalingam and Reps's algorithm. When the algorithm terminates, objects with an infinite depth are unreachable and can be garbage collected.

Due to the dynamic nature of object oriented software, the algorithm must also deal with newly instantiated objects, as they change the heap graph. To

(a) Initial situation. Vertex v_2 is selected as the inconsistent vertex with the lowest key.

(b) Vertex v_2 was made consistent by assigning its rhs to its stored depth. Vertex v_3 is next.

(c) $depth(v_3)$ becomes infinity because it has no predecessors. Vertex v_6 is next.

(d) $rhs(G', v_6)$ is based on v_2 and therefore its stored depth becomes two. No inconsistent vertices left.

(e) Vertices of infinite depth are purged, like v_3. Vertices v_4 and v_5 did not need to be traversed.

Fig. 3. Walkthrough of Ramalingam and Reps's algorithm on graph G' of Figure 2. The vertices in the dashed subgraph are ensured consistent.

ensure that the new object will be seen as reachable from the fictive root, the stored depth of that object must be initialised with infinity. It will then be seen as inconsistent upon the first next run of the algorithm, and as such, it will be made consistent by assigning it with a consistent depth.

3.1   Implementation

An implementation of the algorithm has three main issues to consider, namely (i) how inconsistent vertices are determined, (ii) how an inconsistent vertex with the least key is found and (iii) how predecessors of an object are determined. Algorithm 2 is an implementation of Algorithm 1 for which these issues have been resolved.

(i) The first issue is tackled by lines 3-6 and lines 15-19. Initially, a vertex could be detected on inconsistency by computing its rhs and compare it against its stored depth. Computing the rhs is expensive as it requires the traversal of all the predecessors. Therefore, we first use a safe indication, which we call "dirtyness". A predecessor set is dirty whenever it was modified, like additions and removal of predecessor objects, from the previous predecessor set, $Pred(G, o)$. Also, it

Fig. 4. Organisation of a state in a software model checker. It consists of two threads, each having a callstack. The three objects on the left are root objects, as they are referenced directly from the call stacks.
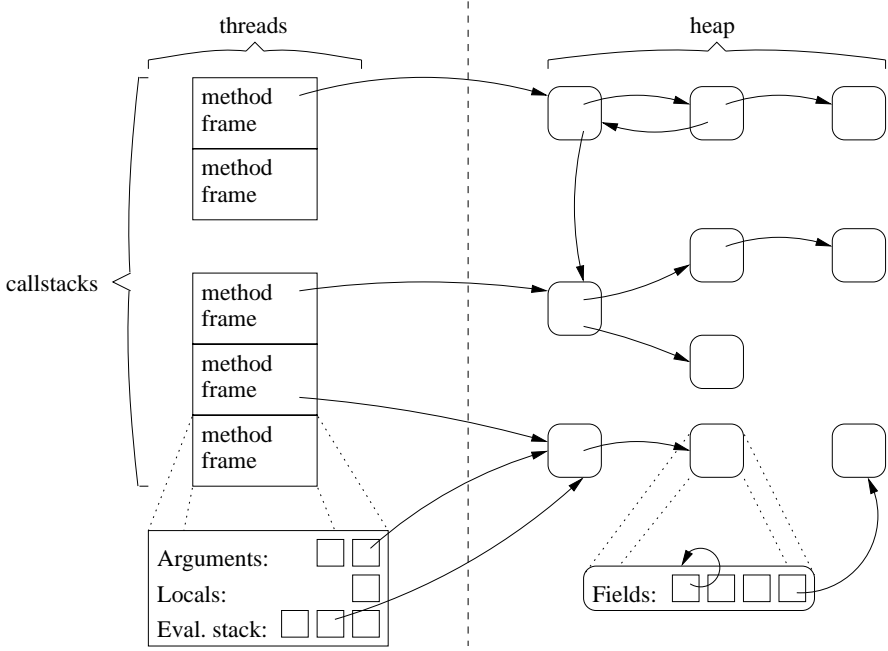
is possible that during one transition, an object is created, used and discarded. Those objects have an empty predecessor set and also have to be considered for inconsistency. When an object passes these tests, then ultimately its rhs is calculated and compared against its depth. Between lines 15-19, successor objects are traversed that could have become inconsistent because their common parent has become consistent. The inconsistent childs are added to the priority queue Q so that they will be made consistent.

(ii) The second issue is determining the object with the least key. Inconsistent objects added to Q are sorted by their key. Due to this order, the object with the least key can be extracted in constant time. In case the key changes of an inconsistent object's that is already in Q (because its predecessor has a changed depth), then this change is reflected by an update to the queue, as done in line 17.

(iii) The third issue relates to function rhs and Algorithm 2. The heap only stores the successor relation explicitly. The predecessor relation can be derived implicitly from this. However, to speed up the algorithm, we maintain an explicit predecessor relation. In MOONWALKER, this relation has to be updated in the following situations:

– Upon interpretation of the `stfld` instruction, if an object reference is stored into an object's field.

Algorithm 2. MemoisedGC(s, s')

Data. priority queue Q
1 $G = (V, E, v_0) \leftarrow$ the object graph associated with state s
2 $G' = (V', E', v_0) \leftarrow$ the object graph associated with state s', the successor of s

3 **foreach** *object $o \in V'$* **do**
4   **if** *$Pred(G', o)$ is dirty $\vee$ $Pred(G', o)$ is empty* **then**
5   **if** $rhs(G', o) \neq depth(o)$ **then**
6   add o to Q with order $key(G', o)$

7 **while** *Q is not empty* **do**
8      $u \leftarrow$ dequeue element from Q with smallest order

9   **if** $rhs(G', u) < depth(u)$ **then**
10          $depth(u) \leftarrow rhs(G', u)$
11          $Affected \leftarrow Succ(G', u)$
12   **else if** $depth(u) < rhs(G', u)$ **then**
13          $depth(u) \leftarrow \infty$
14          $Affected \leftarrow Succ(G', u) \cup \{u\}$

15   **foreach** $o \in Affected$ **do**
16   **if** $rhs(G', o) \neq depth(o)$ **then**
17   **if** $o \in Q$ **then** adjust o on Q with order $key(G', o)$
18   **else** add o to Q with order $key(G', o)$
19   **else if** $o \in Q$ **then** remove o from Q

- Upon interpretation of the `stelem` instruction, if an object reference is stored into an array element.
- Upon a `System.Array.ArrayCopy` internal call, when object references are copied to the destination array.
- When an object reference is pushed on the call stack; then the referenced object becomes a child of the fictive root.
- When an object reference is popped from the call stack; then the referenced object is removed as child of the fictive root.

When state collapse [23] is applied, the predecessor relation also has to be updated similarly upon restoring an object, array and callstack. Furthermore, the predecessor relation has to be stored as a bag (i.e., a counting set). It is possible that an object references another object multiple times by holding the same object reference in multiple fields. If one of these references is removed, then the predecessor relation still holds. The predecessor relation between objects is discarded when all references to the successor object are removed.

3.2   Time Complexity

Whereas the time complexity of M&S is linear to the size of the heap, the time-complexity of MGC is expressed in other terms. The main term is that of an affected object, which is an object whose stored depth has changed during one run of the algorithm. The extended size of an affected object o is $|Pred(o)|$. Given these terms, [19] showed that the worst-case time-complexity of algorithm 2 is

$O(N \cdot (log(N) + M))$, where N is the sum of extended sizes of affected objects plus the amount of affected objects, and M the cost to calculate rhs.

4   Experimental Evaluation

To evaluate the effectiveness of MGC, we wanted to compare it against M&S. M&S was already implemented in MOONWALKER. For running the experiments, we also implemented MGC. It took us three man-months to implement it, including the learning-curve necessary to pick up the .NET platform, to get familiar with MOONWALKER's code and implementing several enhancements to MOONWALKER in between.

4.1   Bandera's Models

Instead of crafting our own benchmarks, we purposely used existing benchmarks. Otherwise one could interpret the results with bias. Thus, we took three models from Bandera's suite, namely `Pipeline`, `SleepingBarbers` and `BoundedBuffer`, and manually ported them to C#. However, after running these academic examples through MOONWALKER, we found them unsuitable for our comparison. The more favourable time-complexity of MGC would only be advantageous for models with big heaps and long verification times. The three small examples have either short verification times (around a second) or very small heaps.

4.2   Java Grande Forum Benchmarks

Benchmarks that resemble real life situations usually have bigger and complex heaps and larger state spaces, making them more interesting and challenging to verify. The three multi-threaded models in the Java Grande Forum Benchmark suite (JGF) [22,24] are such models. These models were developed for the scientific community to evaluate emerging parallel programming paradigms and to expose their weaknesses. Two of these models, MolDyn and Raytracer, were usable. The third benchmark, MonteCarlo, uses file I/O which is not (yet) supported by MOONWALKER. The benchmarks have two parameters, denoted as $t - d$, where t is the number of threads and d is the datasize. For MolDyn, the datasize means the number of particles that is simulated. For Raytracer it means the number of pixels in both width and height that is being rendered. A higher t and/or a higher d will lead to a larger state space. Additionally, to get an idea of the models's size and complexity, its metrics are shown in table 1.

Table 1. Metrics of the MolDyn en Raytracer benchmarks

Metric	MolDyn	Raytracer
#Lines of code	965	1540
#Classes	9	17
#Methods	28	71
#Statements	433	421
#Source code size in Kb.	26	49

As the benchmarks are written in Java, we had to convert them to C#. Due to the size of the code, converting it manually as we did for Bandera's small examples is too error-prone. Instead, we used Microsoft's Java Language Conversion Assistant 3.0, which is included with Microsoft Visual Studio 2005. The conversion was nearly complete and self-contained. The only two things that were not automatically converted were `assert` statements and final field attributes. The first was fixed by manually converting the assert statement to a `System.Diagnostics.Debug.Assert` statement in the resulting C# code. The second was fixed by adding the `readonly` attribute to fields which are marked final in the Java code.

While running initial runs, MOONWALKER found an assertion violation in both models due to a datarace. The datarace occurs over the accesses to variables used to check the assertion and therefore the race does not affect the behaviour of the model. Data races in the Java Grande Benchmarks have also been detected by [6]. While the datarace can be fixed by proper synchronisation of accesses to the concerning variables, we purposely did not do that. We wanted to keep the benchmarks as pure as possible, and secondly, the datarace only increases the state space, so the only side-effect is that the model checker has to do more work.

4.3   Setup

All benchmark runs were performed on a cluster of nine identical systems. Each system has a 2.4 GHz CPU, 2 GB of memory, running Windows XP and installed with .NET 3.0. For both benchmarks, all configurations from 2-1 to 3-3 were ran, with a total of six configurations. Each benchmark run was performed with both static and dynamic partial order reduction enabled [18], a memory threshold of 1.5 GB and a time-limit of 10 hours. A grand total of 24 runs were made, which took a day on the cluster to complete.

4.4   Results

The results of the experiment are summarised in two tables. Table 2 describes the results of the MolDyn benchmark and Table 3 describes the results of the Raytracer benchmark.

The heap size column describes the max. heap size encountered during verification. The time column is the verification time in seconds. A verification that has run out of time is indicated by "o.t.". The memory column is the maximal memory used during verification in megabytes. A verification that has run out of memory is indicated by "o.m.". The states column is the amount of states in the state space. The revisits column is the amount of states revisited during verification. The states stored column is the amount stored in the hashtable. This may differ from the amount of states in the state space due to the ex post facto transition merger that is enabled with stateful dynamic partial order reduction [7]. Note that three columns are represented in thousands for the results from MolDyn benchmarks. The state stored/Mb. column gives an indication of the memory utilisation efficiency. The states/sec. column is the amount of states processed per second during verification. It is calculated by adding the amount of states with the revisits and have that divided by the verification time.

Table 2. MolDyn results with the Memoised Garbage Collector (MGC) and the Mark & Sweep Garbage Collector (M&S)

config.	gc.	heap size (#obj.)	time (sec)	memory (Mb.)	states (·10³)	revisits (·10³)	states stored (·10³)	states stored/Mb	states/sec
2-1	MGC	45	434	1470	1482	1063	1482	1008	5863
	M&S		458	1470	1482	1063	1482	1008	5560
2-2	MGC	101	1447	o.m.	1928	790	978	652	1878
	M&S		1553	o.m.	1926	788	977	651	1748
2-3	MGC	253	78	o.m.	246	0	246	164	3163
	M&S		72	o.m.	249	0	249	166	3475
3-1	MGC	60	913	o.m.	2726	3022	1664	1109	6296
	M&S		1038	o.m.	2724	3018	1662	1108	5531
3-2	MGC	144	91	o.m.	328	0	328	218	3591
	M&S		98	o.m.	327	0	327	218	3324
3-3	MGC	372	153	o.m.	152	0	152	101	993
	M&S		68	o.m.	151	0	151	101	2238

Table 3. Raytracer results with the Memoised Garbage Collector (MGC) and the Mark & Sweep Garbage Collector (M&S)

config.	gc.	heap size (#obj.)	time (sec)	memory (Mb.)	states	revisits	states stored	states stored/Mb	states/sec
2-1	MGC	935	1	37	844	579	844	23	1231
	M&S		1	36	844	579	844	23	1198
2-2	MGC	940	109	664	65923	53264	65923	99	1091
	M&S		113	655	65923	53264	65923	101	1055
2-3	MGC	3254	o.t.	1151	79673	19899	79673	69	3
	M&S		o.t.	1373	97233	24289	97233	71	3
3-1	MGC	1368	38	483	53631	71076	53631	111	3278
	M&S		68	475	53631	71076	53631	113	1842
3-2	MGC	1368	o.t.	1571	32520383	248967	187623	119	910
	M&S		o.t.	1572	30093872	246229	185707	118	843
3-3	MGC	1384	23	o.m.	43330	0	43330	29	1890
	M&S		32	o.m.	43323	0	43323	29	1347

From the table of MolDyn, we observe that MGC is faster (in terms of states/ sec) for configurations 2-1, 2-2, 3-1 and 3-2, with respectively 5%, 7%, 14% and 8% performance increase. The average performance increase with MGC on these configuration is 9%. Table 3 shows that MGC is faster for all configurations except configuration 2-3. The increases are respectively, 3% for both configurations 2-1 and 2-2, 78% for configuration 3-1, 8% for configuration 3-2 and 40% for configuration 3-3. The average performance increase of these configurations is 26%.

We hypothesised that the increase of performance correlates with the heap size. This is partially true. We saw that Raytracer configurations have bigger

heaps, and as such the performance increase is generally higher than those of the MolDyn benchmarks. The latter configurations however revealed a surprising result, namely a huge decline in performance for configuration 3-3 and a moderate decline in performance for configuration 2-3. We investigated this using a profiler and observed that our initial assumption does not always hold. We assumed that the heap does not change much between successive states. This depends however on the heap property that is being measured. The heap shape does not change much, but we did observe that the depth labelling changes much for MolDyn configurations 2-3 and 3-3. As object references are popped and pushed upon the callstacks, the successors of the fictive root change, and thus, also the object graph. Also, these affected objects can cause a chain reaction of changed depth labelling of subsequent successor objects. The MGC bases object reachability on this depth labelling.

Furthermore, the profiler revealed an overhead in the maintenance of parent lists. These list are updated upon every change to the object graph. The changes are especially heavy when a collapsed state is restored, where it is not uncommon that many objects change.

Note that both observations depend on the model that is being verified. The Raytracer model is less susceptible to massive depth-labelling changes between successive states, thereby benefiting more from MGC.

When it comes to memory overhead (in terms of states stored per Mb.), we see that there is no significant difference between MGC and M&S. This means that the memory overhead for maintaining parentlists is neglectible.

5   Future Work

Profiler measurements revealed that MGC decreases the stake of garbage collection from 55% to 24% on Raytracer 3-1. Yet, during development of MGC, we identified several opportunities for further optimising the garbage collection process.

Implementation. The implementation can be further improved by using a HOT queue [3] instead of currently used the Interval Heap for Q in Algorithm 2. The HOT queue has a better time-complexity for monotone increasing keys, which holds for this algorithm. Also the calculation of the *rhs* function can be done in constant-time using the improved algorithm by [19]. Both are more difficult to implement efficiently and for this reason, it is deferred as future work.

No garbage or lots of garbage. While studying the effect of MGC, we observed that lots of calls to the garbage collector do not result in the collection of garbage objects and that some calls result in the collection of lots of objects. The first especially happens when the callstack of a thread grows by successive method calls. If one would develop a method to detect this beforehand and disable the garbage collector, time is saved, especially when combined with M&S. The latter, collection of lots of garbage, occurs when exploration comes closer to the end state. By switching from MGC to M&S, thus a hybrid-approach, would benefit here.

Other incremental shortest path algorithms. The incremental shortest path algorithm by Ramalingam and Reps set off an active field of study on incremental

shortest path calculation. Since its publication, hundreds of publications describing refinements and specialisations have emerged. It is well possible that improvements have been developed that are also applicable to MGC.

Incremental cycle detection with reference counting. The improvements mentioned above are merely to improve the MGC. Our study also gave us an idea for a more fundamental improvement. MGC uses the depth of a vertex as a property to determine reachability. In the end, it is all about the latter, not about the depth. Other properties of a graph might be used instead. For example, a fundamental different approach is to combine reference counting with a form of incremental cycle detection. The incremental cycle detector exploits the changes in a transition by incrementally maintaining the list of cycles in a heap. The reference garbage collector only has to check whether the change causes the cycle to become unreachable from the object graph, and if so, collect the cycle.

Applications of incremental computation. The incremental nature of the MGC is also applicable to other algorithms. For instance, [17] describes an incremental heap canonicalisation algorithm based on Iosif's canonicalisation algorithm [11]. They use the shortest path to achieve this, and, as they suggest themselves, can be calculated incrementally. This can be further extended to gain an incremental k-BOTS algorithm, such that thread symmetries [12] can be detected incrementally.

6   Conclusions

Software model checkers spend around half of their time on garbage collection using the Mark&Sweep algorithm. To optimise this, we describe the Memoised Garbage Collecter, which has a better time-complexity than M&S. In particular, we show how depth-information can be used to determine reachability, how an incremental shortest-path algorithm can be applied to track the depths efficiently, how this algorithm drives the MGC, how this garbage collection algorithm can be implemented and finally an experimental evaluation of it on real-life benchmark models. The performance gain observed from our benchmarks is up to 78% percent, depending on the model and configuration.

Through our work on MGC, we identified several directions for future work (see Section 5) which hopefully lead to more improved garbage collection algorithms and faster software model checking in general.

References

1. Aan de Brugh, N.H.M.: Software Model Checking for Mono. Master's thesis, University of Twente, Enschede, The Netherlands (August 2006)
2. Aan de Brugh, N.H.M., Ruys, T.C., Nguyen, V.Y.: MoonWalker: Verification of .NET Programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 170–173. Springer, Heidelberg (2009)
3. Cherkassky, B.V., Goldberg, A.V., Silverstein, C.: Buckets, Heaps, Lists, and Monotone Priority Queues. In: Saks, M. (ed.) SODA 1997: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, pp. 83–92. Society for Industrial and Applied Mathematics (1997)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Bandera, R.: A Source-Level Interface for Model Checking Java Programs. In: ICSE 2000, pp. 762–765 (2000)

5. Dijkstra, E.: A Note on Two Problems in Connexion with Graphs. In: Numerische Mathematik, vol. 1, pp. 269–271 (1959)
6. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 193–208. Springer, Heidelberg (2006)
7. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model Checking Software. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 110–121. ACM, New York (2005)
8. Grieskamp, W., Tillmann, N., Schulte, W.: XRT- Exploring Runtime for. NET Architecture and Applications. In: Cook, B., Stoller, S., Visser, W. (eds.) Proceedings of the Workshop on Software Model Checking, SoftMC 2005. Electr. Notes Theor. Comput. Sci., vol. 144, pp. 3–26 (2006)
9. Havelund, K.: Java PathFinder, A Translator from Java to Promela. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, p. 152. Springer, Heidelberg (1999)
10. Holzmann, G.J.: The SPIN Model Checker – Primer and Reference Manual. Addison-Wesley, Boston (2004)
11. Iosif, R.: Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In: ASE 2001, pp. 254–261. IEEE Computer Society, Los Alamitos (2001)
12. Iosif, R.: Symmetry Reductions for Model Checking of Concurrent Dynamic Software. STTT 6(4), 302–319 (2004)
13. Iosif, R., Sisto, R.: Using Garbage Collection in Model Checking. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 20–33. Springer, Heidelberg (2000)
14. Jones, R., Lins, R.: Garbage Collection. John Wiley & Sons, Chichester (1996)
15. Lerda, F., Visser, W.: Addressing Dynamic Issues of Program Model Checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)
16. McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM 3(4), 184–195 (1960)
17. Musuvathi, M., Dill, D.L.: An Incremental Heap Canonicalization Algorithm. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 28–42. Springer, Heidelberg (2005)
18. Nguyen, V.Y.: Optimising Techniques for Model Checkers. Master's thesis, University of Twente, Enschede, The Netherlands (December 2007)
19. Ramalingam, G., Reps, T.W.: An Incremental Algorithm for a Generalization of the Shortest-Path Problem. Journal Algorithms 21(2), 267–305 (1996)
20. Robby, Dwyer, M.B., Hatcliff, J.: Domain-specific Model Checking Using The Bogor Framework. In: ASE 2006, pp. 369–370. IEEE Computer Society, Los Alamitos (2006)
21. Ruys, T.C., Aan de Brugh, N.H.M.: MMC: the Mono Model Checker. Electr. Notes Theor. Comput. Sci. 190(1), 149–160 (2007); Proc. of Bytecode 2007
22. Smith, L.A., Bull, J.M., Obdržálek, J.: A Parallel Java Grande Benchmark Suite. In: ACM/IEEE Conference on Supercomputing (SC 2001). ACM, New York (2001)
23. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model Checking Programs. In: ASE 2000, pp. 3–12. IEEE Computer Society, Los Alamitos (2000)
24. The Java Grande Forum Benchmark Suite,
http://www.epcc.ed.ac.uk/research/activities/java-grande/
25. Java PathFinder, http://javapathfinder.sourceforge.net/
26. MoonWalker, http://www.cs.utwente.nl/~ruys/moonwalker/

Hierarchical Adaptive State Space Caching
Based on Level Sampling

Radu Mateescu and Anton Wijs

INRIA / VASY, 655, avenue de l'Europe, F-38330 Montbonnot St Martin, France
{Radu.Mateescu,Anton.Wijs}@inria.fr

Abstract. In the past, several attempts have been made to deal with the state space explosion problem by equipping a depth-first search (DFS) algorithm with a state cache, or by avoiding collision detection, thereby keeping the state hash table at a fixed size. Most of these attempts are tailored specifically for DFS, and are often not guaranteed to terminate and/or to exhaustively visit all the states. In this paper, we propose a general framework of hierarchical caches which can also be used by breadth-first searches (BFS). Our method, based on an adequate sampling of BFS levels during the traversal, guarantees that the BFS terminates and traverses all transitions of the state space. We define several (static or adaptive) configurations of hierarchical caches and we study experimentally their effectiveness on benchmark examples of state spaces and on several communication protocols, using a generic implementation of the cache framework that we developed within the CADP toolbox.

1 Introduction

In model checking, the *state space explosion* problem is the most important issue. It stems from the fact that a linear growth of the number of concurrent processes in a specification leads to an exponential growth of the number of states in the resulting state space. This problem strongly limits the possibilities to verify large systems, since state space generation algorithms typically need to keep all generated states in memory, thereby exhausting it quickly. Over the years, many techniques have been introduced to fight it, e.g. *partial order reduction* [16,5], using *secondary storage* [8,17], *distributed model checking* [13,3,2], and *directed model checking* [9,10].

Another research branch is to consider partial storage of the previously explored states. This idea has lead, roughly speaking, to two classes of approaches: one where exhaustive exploration of the state space is guaranteed, and one where it is not. Since we are concerned with state space *generation*, i.e. the traversal of all the transitions in a state space, we focus on the first class; the reader is referred to Section 5 for the second one. The first class contains most work on state space *caching* [19], where a cache is employed which can never contain more than n states, and a technique which can be referred to as *covering set determination* [1] where, by means of static analysis, the goal is to identify which states to store in order to guarantee termination of the exploration. The caching approach is usually reserved for *depth-first search* (DFS) exploration, since termination can be guaranteed by efficient cycle detection, as opposed to when using

breadth-first search (BFS). However, partial storage of explored states for BFS is desirable as well, since BFS (unlike DFS) can be efficiently distributed in order to perform the state space exploration using clusters of machines.

In this paper, we focus on state space generation with the goal of storing the state space on disk, meaning that besides minimising the memory use, we also aim at reducing the number of state revisits, which determines the amount of redundant information in the generated state space. This distinguishes our work from earlier work on state space caching, where the goal was to visit all states (but not necessarily traverse all transitions) and hence memory use was the main factor of importance. First, we propose a framework allowing hierarchies of caches in addition to single caches. To our knowledge, this has not been investigated yet in this context, although it is quite common practice in the field of hardware architectures [18]. The main idea is that several caches together, each having different characteristics, can be more efficient than one single cache. We study the performance of several hierarchical cache configurations using DFS, some of them improving on results shown by single cache setups. Second, we explain how these caches can be used for BFS by storing *search levels* instead of individual states in them. This provides a generic (language-independent) *on-the-fly* covering set estimation, instead of a (language-dependent) one based on static analysis. Our main result in this setting guarantees termination of the BFS with caches if appropriate *sampling functions* are used to decide which search levels must be stored. Finally, we propose a learning mechanism allowing the BFS algorithm to adapt on-the-fly to the state space structure by detecting its earlier mistakes.

Using the CADP toolbox [12] and the underlying OPEN/CÆSAR environment [11] for state space manipulation, we have implemented generic libraries for hierarchical caches, as well as cache-equipped BFS and DFS exploration tools, which allow to finely tune the memory use. We applied these tools on many examples of state spaces, taken either from benchmarks, or produced from communication protocols. These experiments allowed us to identify the most effective cache configurations, which can lead to memory reductions of over one order of magnitude, and even to speedups of the generation. Our results improve over existing ones by being language-independent and robust to the variations of the so-called *locality* (roughly, the size of back jumps during BFS) of state spaces.

The paper is organized as follows. Section 2 defines the terminology we use for state space exploration. Section 3 presents our cache-based BFS algorithm based on sampling of levels, proves its termination, and defines several instances of it using hierarchical caches and adaptive mechanisms. Section 4 briefly describes the implementation within CADP and gives various performance measures. Finally, Section 5 compares our approach with previous work and Section 6 gives concluding remarks and directions for future work.

2   Preliminaries

Labelled transition systems (LTSs) capture the operational behaviour of concurrent systems. An LTS consists of transitions $s \xrightarrow{\ell} s'$, meaning that being in a state s, an action ℓ can be executed, after which a state s' is reached.

Definition 1. *A* labelled transition system (LTS) *is a tuple* $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, s_0)$*, where* $\mathcal{S}$ *is a set of states,* $\mathcal{A}$ *a set of transition labels,* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *a transition relation, and* s_0 *the initial state. A transition* $(s, \ell, s') \in \mathcal{T}$ *is denoted by* $s \xrightarrow{\ell} s'$.

Furthermore, we express that a state s' is reachable from s with $s \rightarrow^* s'$, where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$. Likewise, we express with $s \rightarrow^+ s'$ that s' is reachable from s by traversing at least one transition. The set of enabled transitions of s is $en(s) = \{(s, \ell, s') \mid \exists \ell \in \mathcal{A}, s' \in \mathcal{S}. s \xrightarrow{\ell} s'\}$, and the set of successor states $succ(s) = \{s' \in \mathcal{S} \mid \exists \ell \in \mathcal{A}. (s, \ell, s') \in en(s)\}$ consists of all states reachable from s via a single transition.

Before generation, the structure of an LTS is not known. A generation algorithm is given an *implicit* LTS $\mathcal{M}_{im} = (s_0, en)$, with s_0 an initial state, and en the enabled function which can be used to generate the other states and the transitions between them, thereby creating an *explicit* LTS in line with Definition 1. We call an LTS *finite* iff $\mathcal{S}$ and $\mathcal{A}$ are of a finite size. Whenever a state is newly discovered, we call it a *visited* state; once we have generated all transitions and successors of s by employing the enabled function, we call s an *explored* state. Recognising when a newly visited state is already explored is called *duplicate detection* (**DD**). In a standard BFS, all previously explored states are stored in memory, in what is called the *Closed* set, and all visited states yet to be explored are stored in the *Open* set, also called the *search horizon*. Given a nonempty *Open* set, the exploration of all the states therein can be seen as an *iteration* of the BFS algorithm, which produces a new *search level* consisting of all the newly generated successors. Subsequently, these states without the duplicates make up the new *Open* set, which can be subjected to a new iteration. **DD** will happen whenever applicable, and the resulting LTS will contain no redundant states, i.e. states which are de facto equal to other states in the LTS. To make things clear when it comes to *partial* **DD**, we represent LTS generation explicitly by first constructing a generation tree of nodes. Figure 1 depicts the generation of an LTS as the traversal of some system behaviour, using both BFS with and without a *Closed* set. Left and right of the behaviour are two trees, where the numbering of the nodes firstly indicates the search level in which the node is encountered, and secondly, in what order the nodes are encountered. The dotted lines visualise which system state each node maps to. Using a *Closed* set, all explored nodes remain in memory, therefore, in the left tree, once nodes 2.4 and 3.5 are visited, they are recognised as essentially being equal to nodes 2.3 and 1.1, respectively, via their associated system states. If we consider BFS without a *Closed* set, on the right of the figure, we observe partial **DD**. Once node 2.4 is encountered, it is recognised as being essentially equal to node 2.3, since 2.3 at that time resides in the *Open* set. However, later on, node 3.5 is not recognised as being equal to node 1.1, since the latter has not been kept in memory, and the generation continues endlessly. From a generation tree, an LTS is derived by creating a state for each node deemed unique. For the right tree in the figure, this results in redundant states.

For checking duplicates, in practice, often a hash table is used to quickly search the contents of *Closed*. Such a hash table uses a hash function $h : \mathcal{S} \rightarrow \mathbb{N}$ to store $h(s)$ for all $s \in Closed$. If h leads to collisions, that is there are $s, s' \in \mathcal{S}$ such that $s \neq s'$ and $h(s) = h(s')$, then *collision detection* is a method to recognise this, which usually means storing multiple states at the same index in a hash array.
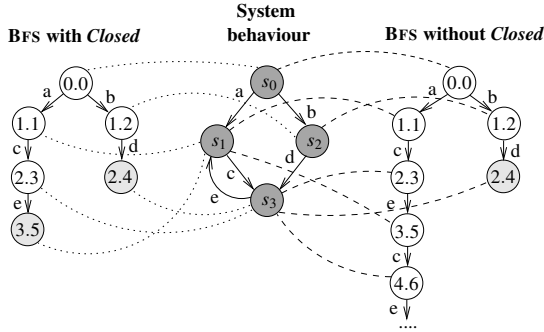
Fig. 1. BFS with and without a *Closed* set

3   BFS with State Space Caching

Caches have a fixed size, and a so-called *replacement strategy*, which determines which element should be removed next, whenever a new element needs to be added to an already full cache. There have been caching attempts with DFS [20,6,27,28,19,15,21,1], and a few with BFS, which, however, are not guaranteed to terminate [29,30].

As with related work on caching with DFS, we intend to restrict the growth of the *Closed* set. This, usually, has a negative effect on two other aspects of LTS generation, namely the execution time and the size of the output LTS. Unlike most previous work, besides the fact that we focus on BFS, we also try to minimise this negative effect. More precisely, we aim at satisfying three criteria: (a) The approach should be independent of any modelling paradigm, that is, it should not rely on analysis of the specification from which the LTS is derived, e.g., by using techniques such as static analysis, as in [1]; (b) The resulting LTS should contain as few duplicates as possible; restriction of the growth of the *Closed* set, however, tends to deteriorate **DD**, unless the LTS has a tree structure, or we are able to store exactly the right states in *Closed*, which would require prior knowledge of the LTS structure; (c) Termination of the generation algorithm must be guaranteed. This is not trivial for BFS, and possibly the main reason that few attempts exist of partial **DD** with BFS; as regards DFS, cycle detection by keeping the DFS stack in memory suffices to guarantee termination.

Without knowledge of the LTS structure, how can we decide which states to store and which to ignore? Many attempts have been made in the past to predict state space structures, e.g. [1,29,7,25], but theoretically, any state may lead to any other state. Some have observed that, although LTSs may have any conceivable structure, the ones stemming from real specifications, which form our main target, seem to share some structural properties [29,7]. Later, we return to this. First, we determine under which conditions the partial storage of explored states in the *Closed* set does not remove the termination guarantee of BFS.

3.1   Partial Storage of Explored States and Termination

Let us consider the relation between partial storage of explored states and termination of BFS. Earlier approaches using BFS all seem to be probabilistic in this respect;

termination is at best highly likely [30,29,7]. Consider the right search tree from Figure 1, now with *Closed* = {1.1}. Note that this is sufficient to generate the LTS without any redundancy. But, focusing on termination, there are more possibilities; consider *Closed* = {2.3}. Even though node 3.5 is not recognised as identical to node 1.1, leading to redundancy in the LTS, the generation will terminate, as node 4.6 will be recognised as identical to the, stored, node 2.3. In other words, the fact that we store search level 2 means that the algorithm terminates. [1] calls a set of vertices in an automaton which ensures termination if states associated with them are stored a *covering set*. Similarly, we call a set of states a covering set if stored related nodes ensure termination. Periodically storing levels seems sufficient to always have a represented covering set in memory. In those cases where a state *s* is explored again, some of its 'descendants', either immediate or remote successors, will be recognised as part of a stored level, hence the redundant work resulting from the detection failure at *s* is finite. The levels should be stored *completely*, otherwise redundant traces may never be recognised as such (more precisely, levels need to be stored without any internal redundancy: in the example tree, note that node 2.4 does not need to be stored, as it is identical to node 2.3). We call these stored levels *snapshots*.

Periodically storing levels allows us the construction of a covering set on-the-fly, as long as there is no bound on the number of snapshots in memory. We investigate this setup in more detail in Section 3.2. However, we are also interested in bounding the number of snapshots in memory, since it allows us to be more rigorous at removing states. Algorithm 1 shows this technique, which we call 'BFS with Snapshots' (BFSWS), where a sampling function $f : \mathbb{N} \to \mathbb{B}$, with $\mathbb{B}$ the domain of the Booleans, is used to determine at which levels to make snapshots, and *n* is the maximum number of snapshots in memory. When $n \to \infty$, we can store an unbounded number of snapshots.

It is important to note that although duplicates are removed from *Next* before the new horizon is created, this is not done when making a new snapshot, in order to keep the levels complete. As explained in Section 4, though, storage of states in snapshots can be implemented such that duplicate occurrences cost little extra memory to keep. Next, Lemma 1 shows under which conditions we can guarantee termination of BFSWS even if the number of snapshots is limited. We prove that BFSWS terminates as long as the sampling period, i.e. the number of levels between the taking of snapshots, increases along the search. From *f*, we can derive a function p_f as follows:

Algorithm 1. BFS with Snapshots

Require: Sampling function $f : \mathbb{N} \to \mathbb{B}$, number of snapshots *n*
  procedure BFSWS(s_0)
    $i, j \leftarrow 0$, *Open* $\leftarrow \{s_0\}$, $S_0, \ldots, S_{n-1} \leftarrow \emptyset$ {Initial state added to horizon}
    $S_j \leftarrow$ *Open*   {First snapshot contains initial state}
    while *Open* $\neq \emptyset$ **do**   {Repeat until there are no more states to explore}
      $i \leftarrow i+1$, *Next* $\leftarrow \emptyset$ {The next level ($i+1$) is currently empty}
      for all $s \in$ *Open* **do**   {Explore all states in the horizon}
        Next $\leftarrow$ *Next* $\cup \{s' \mid \exists \ell.(s,\ell,s') \in en(s)\}$
      Open $\leftarrow$ *Next* $\setminus \bigcup_{k=0}^{n-1} S_k$ {Add new states to horizon}
      if $f(i)$ **then** $j \leftarrow j+1$ mod n, $S_j \leftarrow$ *Next*   {Should this level be sampled?}

- $p_f(0) = 0$, (as we assume $f(0)$ is true)
- $p_f(i) = d$, with $i, d \in \mathbb{N}, i, d > 0$, $f(p_f(0) + \ldots + p_f(i-1) + d) \wedge \forall 0 < d' < d.\neg f(p_f(0) + \ldots + p_f(i-1) + d')$

In words, p_f expresses the subsequent sampling periods observable in f. Now, we need to prove that BFSWS is terminating for finite LTSs if p_f is increasing.

Lemma 1. BFSWS *of a finite* LTS *with a finite number of snapshots $n > 0$, is terminating if p_f is increasing.*

Proof. Let us consider a cycle of size k in an LTS. In a generation tree of BFSWS without DD, this cycle will be generated infinitely often, leading to $s_0, s_1, \ldots, s_{k-1}, s_k \ldots, s_{2k-1}, s_{2k}, \ldots$ etc.[1], with $\forall 0 \leq i \leq k-1, \forall j \in \mathbb{N}.s_i = s_{i+j \cdot k}$. We need to prove that by taking n snapshots, with p_f increasing, the cycle will be detected. First, we consider the case $n = 1$. Other cases follow from this.

Without loss of generality, we say that the first snapshot including a state from the cycle is taken while traversing the cycle for the first time. We call this state $\hat{s}_0 = s_d$, with $0 \leq d \leq k-1$. Let us first consider a p_f with $\forall i \in \mathbb{N}.p_f(i) = p$, and $p < k$. It follows that for subsequent snapshots $\hat{s}_i = s_{d+i \cdot p}$, with $i \in \mathbb{N}$. Observe that with $\hat{s}_0$, DD will succeed when reaching state s_{d+k}. But, since $p < k$, we have $s_{d+p} \rightarrow^+ s_{d+k}$, i.e. $\hat{s}_1 \rightarrow^+ s_{d+k}$. Because $n = 1$, we lose $\hat{s}_0$ after creating $\hat{s}_1$, hence there is no DD when we reach s_{d+k}. Similarly, with $\hat{s}_1$, DD can happen when reaching s_{d+p+k}, but $\hat{s}_2 \rightarrow^+ s_{d+p+k}$. In general, with $\hat{s}_i$, detection may happen at state $s_{d+i \cdot p+k}$, but $\hat{s}_{i+1} \rightarrow^+ s_{d+i \cdot p+k}$. However, if $p \geq k$, we have with $\hat{s}_i$ that $s_{d+i \cdot p+k} \rightarrow^* s_{d+(i+1) \cdot p}$, i.e. $s_{d+i \cdot p+k} \rightarrow^* \hat{s}_{i+1}$, so the next snapshot would be created some time after the exploration of $s_{d+i \cdot p+k}$, but when $s_{d+i \cdot p+k}$ is reached, DD takes place and the cycle traversal is terminated. If p_f is increasing, it follows that there exists $i \in \mathbb{N}$ such that $p_f(i) \geq k$, hence BFSWS can, in that case, deal with any cycle of arbitrary size. The case of BFSWS with $n > 1$ is a generalisation of case $n = 1$. For a p_f with $\forall i \in \mathbb{N}.p_f(i) = p$, if $n \cdot p < k$, for all $\hat{s}_i$, detection may happen at state $s_{d+i \cdot p+k}$, but $s_{d+(i+n) \cdot p} \rightarrow^+ s_{d+i \cdot p+k}$, i.e. $\hat{s}_{i+n} \rightarrow^+ s_{d+i \cdot p+k}$, and $\hat{s}_{i+n}$ replaces $\hat{s}_i$. If $n \cdot p \geq k$, then for $\hat{s}_i$, $s_{d+i \cdot p+k} \rightarrow^* s_{d+(i+n) \cdot p}$, i.e. $s_{d+i \cdot p+k} \rightarrow^* \hat{s}_{i+n}$, hence DD will take place at $s_{d+i \cdot p+k}$. If p_f is increasing, clearly BFSWS with $n > 1$ also terminates.

BFSWS is guaranteed to terminate if the sampling period is bigger than the size of the largest cycle in the LTS. Since we do not know this size a priori, constantly increasing the period ensures that eventually, the sampling period will be big enough. Now that we know under which conditions BFSWS always terminates, we can look at additional techniques to minimise the amount of redundant work, in order to keep the LTSs on disk as small as possible, and the execution time.

3.2   Maximising the Efficiency of Partial Duplicate Detection

BFS With Snapshot Caches. As explained in Section 1, a cache may contain a finite number of elements. For BFSWS, we can use a cache to store snapshots, as opposed to individual states, which is more common in related work. By choosing complete snapshots as elements, BFSWS is guaranteed to terminate, but it is impossible to enforce

---

[1] As we enumerate the states of the cycle here, s_0 is not necessarily the initial state of the LTS.
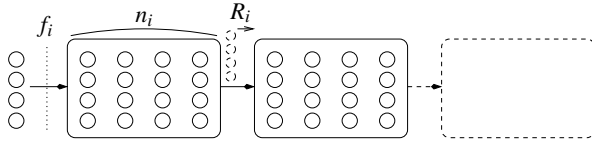
Fig. 2. A stream hierarchy of state space generation caches

a fixed size of the cache. This seems at odds with the principle of a cache, but the caches of webbrowsers work in a somewhat similar manner; such a cache must always contain complete files, not parts of files, even though the files vary in size. For state space caching, this does not cause real problems, since in practice it shows that the gain in memory is still considerable. The size, together with the replacement strategy, which dictates which element to remove in case the cache is full, typically defines a cache. To this, we add the sampling function, which deals with the input of elements, resulting in the following definition.

A *state space generation cache* C_i is a triple (f_i, R_i, n_i), with $f_i : \mathbb{N} \to \mathbb{B}$ the sampling function, $R_i : 2^{2^{\mathscr{S}} \times 2^{\mathbb{N}}} \to 2^{\mathscr{S}}$ the replacement function, taking a set of snapshots together with meta-data about the snapshots (in the form of natural numbers), and returning a snapshot to be removed next, and n_i the maximal number of snapshots in the cache.

Figure 2 illustrates a stream hierarchy of state space generation caches, in a way in which also in hardware architectures, multiple caches can be linked together. The f_i of a cache C_i decides which snapshots to accept for storage, n_i is the maximum number of snapshots it contains, and R_i is the replacement strategy. The removal of a snapshot from C_i leads to the input of a snapshot in C_{i+1}, that is, if f_{i+1} accepts it, etc. In general, R_i computes the cost of every snapshot in the cache based on a cost function $c : 2^{\mathbb{N}} \to \mathbb{N}$, and picks the snapshot with the lowest cost for the next removal. The cost function can use any accumulated data during the generation, e.g., (a) size of the snapshot, (b) snapshot level number, i.e. the time the snapshot was created, (c) the last time **DD** succeeded due to the snapshot, and (d) hit ratio of the snapshot, i.e. how many times **DD** succeeded due to the snapshot.

This machinery allows for a wide range of configurations, since it involves at least four new parameters: the number of caches, and per cache a sampling function, a size, and a replacement strategy. Next, we explain which configurations make sense for the generation of LTSs stemming from real specifications.

Exploiting Transition Locality. Related attempts to search LTSs with partial **DD** often use a notion called *transition locality l* of states [29,7,24]. This is a property of an LTS together with a corresponding traversal tree of *traditional* BFS, i.e. without caching. It expresses the biggest distance, measured in levels, between any two nodes which are considered equal. In Figure 1, for the LTS together with the left tree, $l = 2$, as nodes 1.1 and 3.5 are considered equal. It has been claimed [29,7,24] that l is extremely low for LTSs of real protocol specifications. This can be exploited by only keeping the last l levels of the tree in memory, which yields a version of BFS called *frontier search* [24].[2] Algorithm 1 describes frontier search if $f(i)$ equals *true* for all $i \in \mathbb{N}$,

---

[2] In the original setting of Artificial Intelligence, a predecessor function is assumed to be available to have access to the recent predecessors. Lacking such a function in on-the-fly model checking, the last l levels should be stored [10].

and $n = l$. Termination is only guaranteed if $n \geq l$, $n = l$ being ideal, since it saves as much memory as possible with this technique. However, in practice, l is not known before traversal of the LTS, therefore termination cannot be guaranteed. In addition, our experience points out that, although the majority of equalities concern nodes which are indeed very close to each other, there are usually also some equal nodes much further removed from each other, e.g. more than half the total depth of the tree. This differs from reported results [29,7,24], possibly because we look at a large set of specifications of varying types of systems. In case l is close to the full depth of the tree, which is likely to happen for e.g. cyclic processes, one cannot gain that much memory.

However, the fact remains that often, most nodes which are considered equal are very close to each other in the tree. We choose to exploit this by setting up a stream of caches, the first one sampling frequently, and subsequent ones sampling less often, as identification with 'older' nodes is less likely to happen. Then, for the first cache, we can choose $n_1 < l$, which not only removes the necessity to know l a priori, but for LTSs where l is large, we can also save more memory compared to frontier search and related techniques. Related to this, Tronci et al. [29] deal with distances larger than l in a probabilistic way, while we guarantee eventual detection, hence termination. Streams of caches can be set up in many different ways; we consider two, the results of which will be discussed in the next section:

1. The first cache samples often, in fixed periods, and a second cache employs a sampling function with increasing period, cf. Lemma 1. We call this setup *Frontier Safety Net*, since behind the frontier cache, we can fall back on safety nets.
2. Initially, there is one cache, C_1, sampling often, in fixed periods. As soon as the cache is full, another one, C_2, with the same setup is created and connected behind C_1. Whenever this cache is full, a third one is created, etc. If the sampling period of the caches is not 0, i.e. not every level is sampled, then the further down the stream, the fewer levels are accepted by a cache, hence the longer it takes before the cache is full. Since the number of snapshots allowed in memory is not bounded, this setup guarantees termination of the algorithm. We call this setup *Pebble Search*, since the distribution of the snapshots over a generation tree resembles the waves produced by a pebble when dropped in a pool, i.e. the further away from the point of impact, i.e. the horizon, the further the distance between waves.

The Backtracking Set. In our experience, Frontier Safety Net and Pebble Search lead to good reductions of the memory use, ranging from 50% to sometimes less than 10%, as will be shown in the next section. However, if there are many duplicates to be detected at a distance greater than $n_1 \times p_1$, with p_1 the constant sampling period of cache C_1, then this tends to lead to a big increase of redundant work, negatively affecting both the execution time and the output LTS. Consider Figure 3, where in the *Open* set, node 0 is equal to node 1, which has been explored before, but removed from memory by now. Therefore, node 0 will be re-explored, leading to nodes equal to nodes 2 and 3, the successors of node 1. Since nodes 2 and 3 are also removed, these new nodes are explored as well, and their successors are finally identified as equal to the successors of nodes 2 and 3, since these are present in a snapshot, i.e. the grey bar.
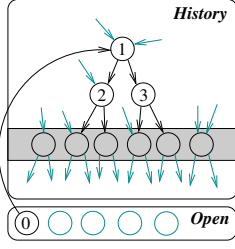
Fig. 3. Duplicate work

All in all, failure to recognise that node 0 has essentially been seen before leads to the traversal of 6 redundant transitions. In state space traversal, the traversal of transitions is the most time-consuming operation, therefore this redundant work has a real impact on the overall execution time. In both our setups, the older the levels, the fewer remain in memory, hence, the larger the distance between equal nodes, the more likely it is that failure of DD leads to the exploration of many nodes before a snapshot is 'reached'. On the one hand, only keeping a few very old levels makes sense, since new nodes do not often refer back to very old nodes. On the other hand, on those occasions where they do, we often obtain a significant amount of redundant work. Let us call the branching factor, i.e. the average number of successors of a state in the LTS, b, and the distance between an old node and the nearest subsequent snapshot d, then every detection failure leads to approximately $\sum_{i=1}^{d} b^i$ additional traversals. In practice, it turns out that if a much older node, i.e. older than only a few levels ago, is referred to again once, then it tends to be referred to several times more later on, in our case each time leading to *at least* $\sum_{i=1}^{d} b^i$ extra traversals (in subsequent re-explorations, the nearest snapshot may well have been removed from memory, thereby increasing the distance to the next snapshot).[3]

These nodes seem to represent states which are very common for the specified systems, imagine e.g. the specification of a car; there are many ways to use the car, but eventually you return to the state representing 'off'. By keeping the node representing 'off' in memory, we can avoid a lot of redundant work. Recognising these important nodes is very hard, but we propose a mechanism for BFSWS which can guess which nodes are important. Every time a node is revisited, the mechanism gets closer to discovering this revisiting. For this we introduce an extra, unbounded, set of nodes to be kept in memory, the *Backtrack* set. While traversing, this set is filled with nodes by following two rules, where very old snapshots are defined as 'not in cache C_1': 1) given a node N in the *Open* set with $| succ(N) | > 1$, if there exists a snapshot S_i not in C_1 such that for all $N' \in succ(N)$, there exists $N'' \in S_i$ with $N' = N''$ (i.e. N' is considered equal to N''), then we add N to *Backtrack*, and 2) given a node N in the *Open* set with $| succ(N) | > 1$, if for all $N' \in succ(N)$, there exists $N'' \in Backtrack$ with $N' = N''$, then we add N to *Backtrack*. The first rule states that if all the successors of a node are detected as duplicates due to a single very old snapshot, then it is very likely that we have explored their parent before. The more successors the node has, the more likely this is, hence we exclude the case here of a single successor. Failure to detect duplicates which only have one successor does not directly lead to much redundant work anyway. The second rule is a continuation of this: if all the successors of a node are suspected of having been re-explored, then we suspect this node as well. With this technique, we bound and lessen the amount of redundant work with each revisit of a node; the nth revisit leads to $\sum_{i=0}^{|d-(n-1)|} b^i - 1$ extra traversals. Practice shows that this learning mechanism is very

---

[3] [19] reports that no relation is found between the number of previous visits to a state and the likelihood that it will be visited again in the future. We found that revisits are likely to states which had 'late' revisits, i.e. revisits many levels after the first visit.

successful, as seen next. By only keeping a few extra nodes in memory according to these rules, we sometimes reduce the amount of redundant work considerably.

## 4   Implementation, Caching Setups, and Experiments

We built a generic, application-independent implementation of the caching machinery using the OPEN/CÆSAR [11] environment of the CADP toolbox [12], which provides various primitives for state space storage and exploration (hash tables, edge lists, stacks, etc.). The cache library allows to define caches containing a fixed number of elements, either states or snapshots, each one being possibly assorted with user-defined information allowing, e.g., to calculate the cost associated to the element. The replacement strategy used by a cache can be user provided, or selected among five built-in strategies: least/most recently used (LRU/MRU), least/most frequently used (LFU/MFU), and random (RND). The elements of a cache are stored in a balanced heap equipped with a hash table in order to allow fast retrievals of the lowest-cost element and fast searches of elements. A special primitive retrieves the last element replaced after an insertion took place in an already full cache; this allows to manage hierarchical caches (organized, e.g., as trees) by retrieving an element from one cache and inserting it into another.

The most basic usage of the cache library is for storing visited states, assorted with their id's, during a DFS traversal of the state space. A more complex usage is e.g. for BFSWS, where elements stored in the cache are snapshots, but the searches carried out for **DD** concern individual states. To reduce memory consumption, states belonging to the set of snapshots currently present in a cache are stored uniquely and referenced through pointers; state deletion is done efficiently using a reference counting scheme borrowed from garbage collection [22]. Next, we study the performance of several BFS and DFS setups experimentally. For this, we used around 35 LTSs from the VLTS benchmark suite stemming from real, industrial case studies (http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html) and also several communication protocols (http://www.inrialpes.fr/vasy/cadp/demos.html). The experiments were run on a LINUX machine with a 2.2GHz CPU and 1 GB memory.

4.1   BFS Experiments

Here, we show the results of a representative selection of cases, generated by standard BFS and by BFSWS using some of the most successful cache setups. Caches are described as triples, a sampling function which increases its period by n after every sampling being written as n, a function with constant period n being written as cn, LLNR being a replacement function based on lowest level number, and BT indicating the backtracking mechanism. The top three graphs visualise the results on several instances of the Bounded Retransmission Protocol (BRP), varying in both the size of messages and the number of retransmissions. Here, the techniques are extremely effective, allowing not only to reduce the memory use drastically, but also to make the generation much faster. This is due to the hash table being very small, which speeds up **DD**. Usually, this gain is countered by failed detections, leading to more (time consuming) transition traversals, but here such failures hardly occur. Additional tests showed that we could

generate the LTS for BRP $\langle 300, 300 \rangle$, consisting of more than $410,000,000$ states, in 37 hours with $(c2, \text{LLNR}, 5)(2, \text{LFU}, 10)$, and in 22 hours with backtracking. The bottom three graphs show the generation results for the SCSI-2 bus arbitration protocol, varying the number of competing disks. Here, memory use can be reduced to 20% compared to standard BFS, using a Frontier Safety Net with backtracking. Moreover, both backtracking setups show practically no increase in execution time, and the number of redundant states produced is reasonable.
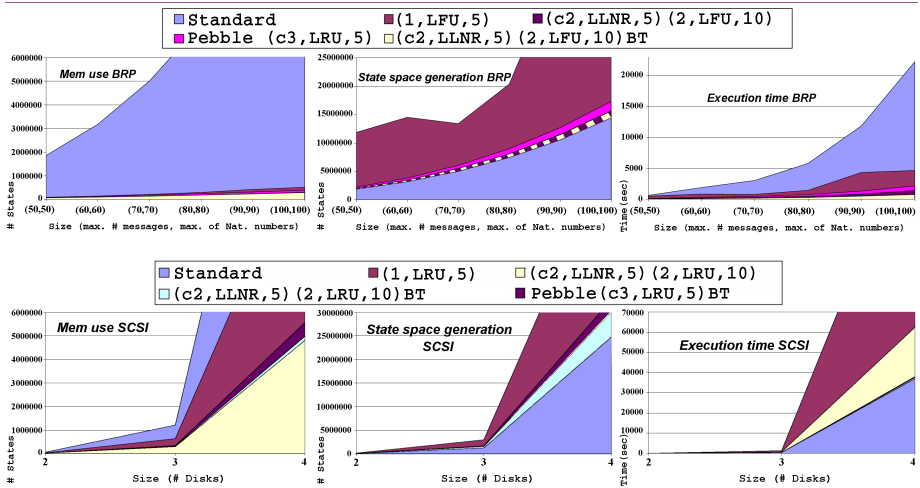

The graphs below compare execution times and state generation of BFS and 3 sampling setups. Execution times of the setups are often much longer than for BFS, due to using more complex data structures, and the redundant work. However, this effect becomes mainly apparent with small examples. The bottom two graphs relate output LTS sizes with the original sizes, indicating the number of states in memory by the size of the bubbles. Bubbles on the same horizontal line relate to the same case. Pebble Search with backtracking sometimes produces remarkably smaller LTSs than the other methods, and faster, suggesting that for these cases, keeping snapshots which are more evenly distributed over the history of the generation pays off. In Frontier Safety Net, the second cache samples more and more infrequently, eventually leaving a big 'hole' in the stored history. There is still room for improvement, though, which we plan as future work. For most cases, memory use can be reduced to about 30% using the sampling mechanism. The graphs show that either backtracking has a very positive effect (e.g. in execution time), or no effect; a negative effect hardly ever occurs. This makes backtracking a useful feature to enable by default in the BFSWS state space generator.
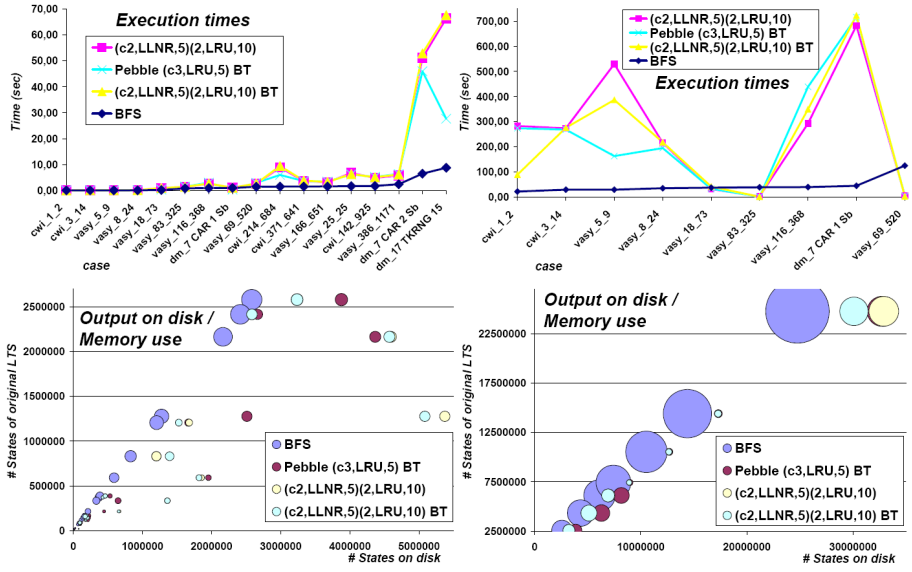
4.2 DFS with Caches

Our generic cache machinery can also be used in conjunction with other graph traversals to reduce the memory needed for LTS generation. The figure aside shows the behaviour of DFS equipped with cycle detection (by searching states on the DFS stack) and a hierarchical $\langle$MFU, LRU$\rangle$ cache, executed on a VLTS subset. For each example, the

figure gives the minimal size of the cache yielding an output LTS of size at most double w.r.t. the input LTS. Among the five built-in replacement strategies, LRU performs best on all examples (reducing the cache size down to 40% of the number of states), followed closely by MFU and RND. LRU is close to the strategy removing the oldest states, rated best among the strategies analysed in [19]. If we split the cache into two cascading subcaches of varying sizes and different strategies, the best cache size reduction (down to 30% of the number of states) was achieved by using an MFU or RND subcache followed by an LRU one (of about 25% of the whole cache). Overall performance (see the figure) was further improved by increasing the LRU subcache to 75%. Compared to BFSWS, the success of DFS setups differs a lot from one case to another. Moreover, a difficulty with "fixed size" DFS caching is to determine the right size of the cache, which, in order to be effective, should lie between $30 - 60\%$ of the (*a priori* unknown) LTS size; BFSWS, on the other hand, simply takes whatever memory it needs.
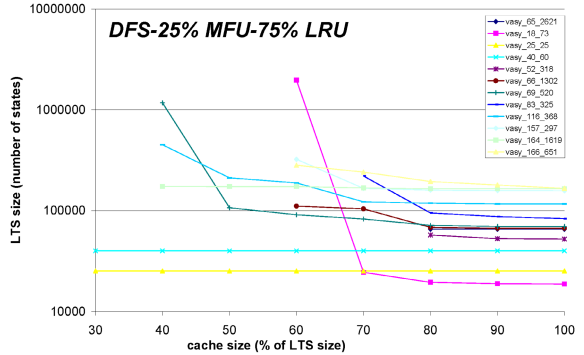
5 Related Work

As mentioned in the introduction, existing approaches for state space generation can roughly be divided in two classes. In the second class, where exhaustiveness is not guaranteed, hashing without collision detection is often used, which is a way to ensure that the hash table stores a fixed number of states. Concerning collisions, [20,6] assume that whenever a collision happens in the hash table, the new state has already been visited. In [6], this is used for a nested DFS to search for errors in LTSs. Collision avoidance in this way guarantees termination. [29,7] take the opposite approach concerning this. They fully depend on LTSs of protocols having small localities. They avoid collision detection, by removing a previously stored state if there is a hash collision with a new state, arguing that in most cases this means removing a state beyond the locality region. Termination is handled by stopping the search once the collision rate is sufficiently high, or a maximum search depth has been reached. In [30], a fixed size cache and *Open* set is used for a probabilistic, randomised BFS, where states are replaced at random. These last two cases report memory savings of 40% on average, the first case achieving this partly by keeping the *Open* set on disk. In [27], the probability of failures is reduced by using open addressing and t-limited lookups and insertions in the hash table, where a hash function provides not one position for a state in the hash table, but a sequence of t possible locations. In addition to this, [28] includes the level number of a state in a BFS in calculating its omission probability.

The other class, guaranteeing exhaustiveness, includes most state space caching work, which has been done for DFS [19,15,21,14]. In [19], several replacement strategies are used for a single LTS, and the conclusion is that the one selecting the oldest states is the best. [21] continue on this, believing the conclusion to be a random strategy. [14] reinvestigates this, employing many other strategies on many examples, some of them from practical cases. In those cases, a proposed strategy called stratified caching with random replacement performs best. Here, *strata* are created, very similar to our snapshots. The states of certain strata, however, are candidates for *removal* from memory, as opposed to storing in memory. Caching is combined with static analysis and partial order reduction in [15]. Our setting allows hierarchies of caches, and they can be used to store snapshots in case of BFS. In [1], an unbounded set of states is stored based on a storing strategy, usually based on static analysis. They claim that up to 90% can be saved in memory use, but, as reported by [17], this leads to an extensive amount of redundant work. In [17], a hash table is initially used in a traditional manner, until it fills up the whole memory, at which point older states are moved to secondary storage. The number of lookups on disk is reduced by using a Bloom filter. Instead of trying to use memory in a smarter way, they want the generation to be able to continue once the memory is filled. We experience that trying to avoid a big *Closed* set pays off in terms of execution time, since the hash table is kept small. However, our BFS method may still run out of memory. It could be interesting to look at a combination of the two.

In Artificial Intelligence (AI), connecting to directed model checking, multiple methods are presented to bound the memory use of exhaustive search algorithms, e.g. *IDA** [23], *MA** [4], and extensions *IE* and *SMA** [26]. As is common in this field, the algorithms employ a cost function, mapping states to costs. For memory-bounding, this function is used to decide which states to remove from memory. The cost function

provides knowledge of the LTS structure a priori, and its main purpose is to guide the search to a goal state. In this paper, we are neither concerned with a subset of goal states, nor have any structural knowledge. It is, however, possible to incorporate the AI algorithms in our framework, like DFS and BFS, in order to obtain more memory efficient variants.

6 Conclusion and Future Work

We presented generic machinery for hierarchical, adaptive state space caching, implemented using the OPEN/CÆSAR environment [11] of the CADP toolbox [12]. This machinery can be used in a very flexible manner for state space generation, in conjunction with DFS and BFS traversals. Our algorithm BFSWS is exhaustive and guaranteed to terminate, and its behaviour can be finely tuned using the caching and learning mechanisms introduced in Section 3. These techniques compete favourably with earlier ones, such as hashing without collision detection and frontier search, which only concern LTS search (and not generation), lack termination or exhaustiveness (or both), or are dedicated to LTSs with small localities. The learning mechanism for BFSWS strongly reduces the amount of redundant work, and is also able to speed up the generation.

As future work, we will study other BFSWS configurations on further examples (e.g. the BEEM benchmark [25]), and try to design additional mechanisms to deal more efficiently with different LTS structures. Secondly, we plan to adapt the machinery for distributed state space generation [13]. Finally, we want to investigate its use in conjunction with on-the-fly LTS reduction modulo τ-confluence and branching bisimulation.

References

1. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
2. Blom, S., Calamé, J.R., Lisser, B., Orzan, S., Pang, J., van de Pol, J., Dashti, M.T., Wijs, A.J.: Distributed analysis with μCRL: A compendium of case studies. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 683–689. Springer, Heidelberg (2007)
3. Blom, S.C.C., Orzan, S.: Distributed State Space Minimization. STTT 7(3), 280–291 (2005)
4. Chakrabarti, P.P., Ghose, S., Acharya, A., De Sarkas, S.C.: Heuristic Search in Restricted Memory. Artificial Intelligence 41(2), 197–222 (1989)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
6. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-Efficient Algorithms for the Verification of Temporal Properties. FMSD 1(2–3), 275–288 (1992)
7. Della Penna, G., Intrigila, B., Tronci, E., Venturini Zilli, M.: Exploiting Transition Locality in the Disk Based Murphi Verifier. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 202–219. Springer, Heidelberg (2002)
8. Edelkamp, S., Jabbar, S.: Real-time model checking on secondary storage. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 67–83. Springer, Heidelberg (2007)
9. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed Explicit-State Model Checking in the Validation of Communication Protocols. STTT 5(2–3), 247–267 (2004)

10. Edelkamp, S., Schuppan, V., Bošnački, D., Wijs, A.J., Fehnker, A., Aljazzar, H.: Survey on Directed Model Checking. In: Peled, D., Wooldridge, M. (eds.) MoChArt 2008. LNCS (LNAI), vol. 5348, pp. 65–89. Springer, Heidelberg (2009)
11. Garavel, H.: OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
12. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
13. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 217–234. Springer, Heidelberg (2001)
14. Geldenhuys, J.: State caching reconsidered. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 23–38. Springer, Heidelberg (2004)
15. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-Space Caching Revisited. FMSD 7(3), 227–241 (1995)
16. Godefroid, P., Wolper, P.: Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 410–429. Springer, Heidelberg (1992)
17. Hammer, M., Weber, M.: ”To Store or Not To Store” Reloaded: Reclaiming Memory on Demand. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
18. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann, San Francisco (2006)
19. Holzmann, G.J.: Automated Protocol Validation in Argos, assertion proving and scatter searching. IEEE Trans. on Software Engineering 13(6), 683–696 (1987)
20. Holzmann, G.J.: An Improved Protocol Reachability Analysis Technique. Software - Practice and Experience 18(2), 137–161 (1988)
21. Jard, C., Jéron, T.: Bounded-memory Algorithms for Verification On-the-fly. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 192–202. Springer, Heidelberg (1992)
22. Knuth, D.E.: The Art of Computer Programming — Sorting and Searching. Computer Science and Information Processing, vol. III. Addison-Wesley, Reading (1973)
23. Korf, R.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence 27(1), 97–109 (1985)
24. Korf, R., Zhang, W., Thayer, I., Hohwald, H.: Frontier Search. J. ACM 52(5), 715–748 (2005)
25. Pelánek, R.: Properties of state spaces and their applications. STTT 10(5), 443–454 (2008)
26. Russell, S.: Efficient memory-bounded search methods. In: Neumann, B. (ed.) ECAI 1992, pp. 1–5. Wiley, Chichester (1992)
27. Stern, U., Dill, D.L.: Combining State Space Caching and Hash Compaction. In: 4. GI/ITG/GME Workshop, pp. 81–90. Shaker Verlag, Aachen (1996)
28. Stern, U., Dill, D.L.: A New Scheme for Memory-Efficient Probabilistic Verification. In: Gotzhein, R., Bredereke, J. (eds.) FORTE 1996, pp. 333–348. Chapman and Hall, Boca Raton (1996)
29. Tronci, E., Della Penna, G., Intrigila, B., Venturini Zilli, M.: Exploiting Transition Locality in Automatic Verification. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 259–273. Springer, Heidelberg (2001)
30. Tronci, E., Della Penna, G., Intrigila, B., Venturini Zilli, M.: A Probabilistic Approach to Automatic Verification of Concurrent Systems. In: APSEC 2001, pp. 317–324. IEEE Press, New York (2001)

Static Analysis Techniques for
Parameterised Boolean Equation Systems[*]

Simona Orzan, Wieger Wesselink, and Tim A.C. Willemse

Eindhoven University of Technology, The Netherlands

Abstract. Parameterised Boolean Equation Systems (PBESs) can be used to encode and solve various types of model checking and equivalence checking problems. PBESs are typically solved by symbolic approximation or by instantiation to Boolean Equation Systems (BESs). The latter technique suffers from something similar to the state space explosion problem and we propose to tackle it by static analysis techniques, which we tailor for PBESs. We introduce a method to eliminate redundant parameters and a method to detect constant parameters. Both lead to a better performance of the instantiation and they can sometimes even reduce problems that are intractable due to the infinity of the underlying BES to tractable ones.

1 Introduction

Model checking and equivalence checking techniques are very sensitive to the size of the state space. A *static analysis* can be used to reduce the state space size; most often, it employs some form of flow analysis to detect what values a given subexpression of a process description can possibly evaluate to at run-time [9], and this information can subsequently be used to achieve state space reductions. A further minimisation might be obtained if the analysis is tailored to the properties to be verified. This constitutes a major challenge, as it requires analysing both the verification question and the specification. One can avoid this by encoding the verification problem in a single high-level formalism; *Parameterised Boolean Equation Systems* [11,7] allow to do just that.

Parameterised Boolean Equation Systems (PBESs) have emerged as a versatile framework for studying and solving verification problems. Prime examples are the PBES encoding of the first-order modal μ-calculus model checking problem over (possibly infinite) labelled transition systems [11,7] and equivalence checking of various bisimulations on (possibly infinite) labelled transition systems [1]. Intuitively, the PBES encoding of a given verification problem only requires the aspects of the specification that influence the property that has to be verified.

Problems encoded in the PBES framework can be solved by computing the solution to the respective PBES. Even though the latter is an undecidable problem, a number of techniques have been developed to obtain solutions in practice, including *symbolic approximation* [7], *pattern matching* [8], *invariant* techniques [13] and *instantiation* [11,12,4].

In this paper, we are concerned with the latter, i.e. instantiation of PBESs to Boolean Equation Systems (BES); BESs constitute a decidable fragment of PBESs.

We develop two methods, based on static analysis, that allow to automatically reduce the complexity of a PBES. These methods take inspiration from [6], where comparable methods are applied to a symbolic state space description. We first investigate, and prove the correctness of a method that allows to eliminate a class of redundant data parameters from a PBES; second, we develop an algorithm, and prove its correctness, that computes a special type of PBES invariant [13], which can subsequently be used to eliminate data parameters and simplify the right-hand sides of a PBES.

The practical significance of the two complexity reduction methods is assessed by means of a set of experiments derived from typical model checking problems. These demonstrate dramatic improvements in the time needed to compute a BES from a PBES, and the size of these BESs; some intractable problems even reduce to tractable ones.

A third contribution of our paper is the introduction of a *normal form* for PBESs. The existence of the normal form has both theoretical and practical implications. On the one hand, it allows for more concise definitions and a uniform presentation of manipulation methods, and, on the other hand, it will help to find, e.g., new patterns [8]. Apart from using the normal form in our definitions of the two mentioned complexity reduction methods, we also use it to simplify the characterisation of invariants for PBESs, paving the way for automating the detection and checking of complex invariants.

Related Work. As mentioned, we take inspiration from [6], where similar static analysis techniques are applied to reduce state spaces. Other forms of static analysis techniques include abstract interpretation, initiated in [3] and used in, e.g. [10,14], influence analysis in programming languages [5], and the so-called *cone of influence reduction* (also known as *slicing* or *localisation reduction*) technique that reduces the size of the state space for synchronous circuits in specific (see [2]) and systems in general (see [16]). Compared to these works, our methods deal with a more advanced setting and have the potential to immediately (and soundly) reduce the complexity of (encoded) verification problems. As such, we can solve verification questions that cannot readily be answered by only reducing the state space (see e.g. our Example 1).

Outline. In Section 2 the basic PBES theory is repeated. Section 3 describes a normal form for PBESs and its implications for invariants. The two complexity reduction methods are described in Section 4, and an analysis of the impact of these algorithms on typical model checking problems can be found in Section 5. Section 6 summarises the main results of this paper and discusses future work.

2   Preliminaries

2.1   Data

We work in the setting of *abstract data types*, i.e., we assume that there are nonempty data sorts and operations on these sorts. We typically use letters $D_1, D_2, \ldots$ to denote data sorts. Furthermore, we assume to have a set $\mathcal{D}$ of *sorted data variables*, with typical elements $d, d_1, \ldots$, *etcetera*. We write $\boldsymbol{d}$, which stands for a vector of variables $(d_1, \ldots, d_n)$; this notation extends to vectors of terms $\boldsymbol{e}$, vectors of sorts $\boldsymbol{D}$ and vectors

of values v in the semantic domain. A vector of sort declarations $d{:}D$ should be read as $(d_1{:}D_1, \ldots, d_n{:}D_n)$. The i-th element of d is denoted $d[i]$.

With every syntactic sort D we associate a semantic set $\mathbb{D}$ such that every syntactic term of type D and all the operations on the sort D can be mapped to the elements and operations of $\mathbb{D}$ they represent. For the interpretation of closed data terms, we assume an interpretation function $[\![\,]\!]$ that maps every term t of sort D to the data element $[\![t]\!]$ of $\mathbb{D}$ it represents. For open terms we use an *environment* ε that maps each variable from $\mathcal{D}$ to a data element of the associated type. The interpretation $[\![t]\!]\varepsilon$ of an open term t is given by $\varepsilon(t)$, where ε is extended to terms in the standard way.

For arbitrary environment θ, we write $\theta[v/d]$ to represent the environment that is defined as $(\theta[v/d])(d') = \theta(d')$ for $d \neq d'$ and $(\theta[v/d])(d) = v$. For substitution on vectors we define $\theta[v/d]$ to be equivalent to the simultaneous substitution $\theta[v[1]/d[1], \ldots, v[n]/d[n]]$.

For convenience, we assume the existence of a sort $B = \{\top, \bot\}$ representing the Booleans $\mathbb{B}$. For this sort, we assume the usual operators are available and we do not write constants or operators in the syntactic domain any different from their semantic counterparts. For example, we have $\mathbb{B} = \{\top, \bot\}$ and the syntactic operator $\_ \wedge \_{:}B \times B \to B$ corresponds to the usual, semantic conjunction $\_ \wedge \_{:}\mathbb{B} \times \mathbb{B} \to \mathbb{B}$.

2.2   Parameterised Boolean Equation Systems

Parameterised Boolean Equation Systems (PBESs, or *equation systems* for short) are empty (denoted ϵ) or finite sequences of fixed point equations, where each equation is of form $\big(\mu X(d{:}D) = \phi\big)$ or $\big(\nu X(d{:}D) = \phi\big)$. The left-hand side of each equation consists of a *fixed point symbol*, where μ indicates a least and ν a greatest fixed point, and a *sorted predicate variable* X of sort $D \to B$, taken from some countable domain of sorted predicate variables $\mathcal{X}$. The right-hand side of each equation is a *predicate formula* as defined below.

Definition 1. Predicate formulae ϕ *are defined by the following grammar:*

$$\phi ::= b \mid X(e) \mid \phi \oplus \phi \mid \mathsf{Q}d{:}D.\ \phi$$

where $\oplus \in \{\wedge, \vee\}$, $\mathsf{Q} \in \{\forall, \exists\}$, b *is a data term of sort* B, X *is a predicate variable, d is a data variable of sort D and e is a vector of data terms. The interpretation of ϕ in the context of environments η for predicate variables and ε for data variables is denoted* $[\![\phi]\!]\eta\varepsilon$, *where:*

$$
\begin{aligned}
[\![b]\!]\eta\varepsilon &=_{def} \varepsilon(b) & [\![\phi_1 \oplus \phi_2]\!]\eta\varepsilon &=_{def} [\![\phi_1]\!]\eta\varepsilon \oplus [\![\phi_2]\!]\eta\varepsilon \\
[\![X(e)]\!]\eta\varepsilon &=_{def} \eta(X)(\varepsilon(e)) & [\![\mathsf{Q}d{:}D.\ \phi]\!]\eta\varepsilon &=_{def} \mathsf{Q}v \in \mathbb{D}.\ [\![\phi]\!]\eta\varepsilon[v/d]
\end{aligned}
$$

We denote the freely occurring data variables in a formula ϕ by $\mathcal{FV}(\phi)$. In line with [13], predicate formulae that do not contain predicate variables are called *simple* predicate formulae; Pred is the set of simple formulae. A simple predicate formula ϕ satisfies the property $[\![\phi]\!]\eta\varepsilon = [\![\phi]\!]\eta'\varepsilon$, for arbitrary η, η'. As a convention, we denote simple predicate formulae using letters g, h, *etcetera*. Observe that negation does not occur in predicate formulae, except as an operator in Boolean terms. We frequently write $h \implies \phi$ instead of $\neg h \vee \phi$.

The set of predicate variables that occur in a predicate formula ϕ, denoted by $\mathsf{occ}(\phi)$, is defined recursively as follows, for any formulae ϕ_1, ϕ_2:

$$\mathsf{occ}(b) =_{def} \emptyset \qquad\qquad \mathsf{occ}(X(\boldsymbol{e})) =_{def} \{X\}$$
$$\mathsf{occ}(\phi_1 \oplus \phi_2) =_{def} \mathsf{occ}(\phi_1) \cup \mathsf{occ}(\phi_2) \qquad \mathsf{occ}(\mathsf{Q}d{:}D.\ \phi_1) =_{def} \mathsf{occ}(\phi_1).$$

Extended to equation systems, $\mathsf{occ}(\mathcal{E})$ is the union of all variables occurring at the right-hand side of equations in $\mathcal{E}$. For any equation system $\mathcal{E}$, the set of *binding predicate variables*, $\mathsf{bnd}(\mathcal{E})$, is the set of variables occurring at the left-hand side of some equation in $\mathcal{E}$. Formally, we define:

$$\mathsf{bnd}(\epsilon) =_{def} \emptyset \qquad \mathsf{bnd}((\sigma X(\boldsymbol{d}{:}\boldsymbol{D}) = \phi)\ \mathcal{E}) =_{def} \mathsf{bnd}(\mathcal{E}) \cup \{X\}$$
$$\mathsf{occ}(\epsilon) =_{def} \emptyset \qquad \mathsf{occ}((\sigma X(\boldsymbol{d}{:}\boldsymbol{D}) = \phi)\ \mathcal{E}) =_{def} \mathsf{occ}(\mathcal{E}) \cup \mathsf{occ}(\phi).$$

The set of freely occurring predicate variables in $\mathcal{E}$, denoted $\mathsf{free}(\mathcal{E})$ is defined as $\mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$. An equation system $\mathcal{E}$ is said to be *well-formed* iff every binding predicate variable occurs at the left-hand side of precisely one equation of $\mathcal{E}$. We only consider well-formed equation systems in this paper.

An equation system $\mathcal{E}$ is called *closed* if $\mathsf{free}(\mathcal{E}) = \emptyset$ and *open* otherwise. An equation $(\sigma X(\boldsymbol{d}{:}\boldsymbol{D}) = \phi)$, where σ denotes either the fixed point sign μ or ν, is called *data-closed* if the set of data variables that occur freely in ϕ is contained in the set of variables induced by the vector of variables $\boldsymbol{d}$. An equation system is called *data-closed* iff each of its equations is data-closed.

An equation $(\sigma X(\boldsymbol{d}{:}\boldsymbol{D}) = \phi)$ gives rise to a fixed point over the set of functions with domain $\mathbb{D}$ and co-domain $\mathbb{B}$. We introduce the notation $\phi_{\langle \boldsymbol{d} \rangle}$, which lifts the predicate formula ϕ to the (syntactic) functional $(\lambda \boldsymbol{d}{:}\boldsymbol{D}.\ \phi)$. The interpretation of $\phi_{\langle \boldsymbol{d} \rangle}$, denoted $[\![\phi_{\langle \boldsymbol{d} \rangle}]\!] \eta \varepsilon$, is given by the functional $(\lambda \boldsymbol{v} {\in} \mathbb{D}.\ [\![\phi]\!] \eta \varepsilon [\boldsymbol{v}/\boldsymbol{d}])$. The set of (total) functions $f{:}\mathbb{D} \to \mathbb{B}$, denoted $\mathbb{B}^{\mathbb{D}}$, equipped with the point-wise ordering $\sqsubseteq$ leads to a complete lattice. Assuming that the domain of the predicate variable X is of sort $\boldsymbol{D}$, the functional $[\![\phi_{\langle \boldsymbol{d} \rangle}]\!] \eta \varepsilon$ yields the monotone predicate formula transformer $\lambda g {\in} \mathbb{B}^{\mathbb{D}}.\ ([\![\phi_{\langle \boldsymbol{d} \rangle}]\!] \eta [g/X] \varepsilon)$. The existence of the least and greatest fixed points of such transformers is guaranteed by Tarski's fixed point Theorem [15].

Definition 2. *The* solution of an equation system *in the context of a predicate environment η and a data environment ε is inductively defined as follows, for any $\mathcal{E}$:*

$$[\![\epsilon]\!] \eta \varepsilon =_{def} \eta$$
$$[\![(\sigma X(\boldsymbol{d}{:}\boldsymbol{D}) = \phi)\mathcal{E}]\!] \eta \varepsilon =_{def} [\![\mathcal{E}]\!] (\eta [\sigma f \in \mathbb{B}^{\mathbb{D}}.\ [\![\phi_{\langle \boldsymbol{d} \rangle}]\!] ([\![\mathcal{E}]\!] \eta [f/X] \varepsilon) \varepsilon / X]) \varepsilon.$$

The solution of an equation system prioritises the fixed point signs of equations that come first over the signs of equations that follow, while respecting the equivalences of the equations. It follows that the solution is sensitive to the order of equations in an equation system. We illustrate the use of equation systems by means of an academic example using the encoding of [7] of the first-order modal μ-calculus model checking problem.

Example 1. Consider an infinite-state process that can perform an arbitrary number of a actions, then performs a b action and then performs as many c actions as a actions that were performed. A partial visualisation of this process, and a process algebraic description using condition-action-effect rules is given below:

$$P(n{:}N, d{:}B)$$
$$= d \longrightarrow a \cdot P(n+1, d)$$
$$+ d \longrightarrow b \cdot P(n, \neg d)$$
$$+ \neg d \wedge n > 0 \longrightarrow c \cdot P(n-1, d)$$

Given this process, we might wish to verify whether it is possible to perform an infinite number of a actions ($\nu X.\langle a \rangle X$), or, whether along every a path, always a b action is attainable ($\nu V.\ ([a]V \wedge \mu W.\ (\langle a \rangle W \vee \langle b \rangle \top))$). The first verification problem is encoded by (1); the second by (2), given below:

$$\big(\nu X(n{:}N, d{:}B) = d \wedge X(n+1, d) \big) \tag{1}$$

$$\big(\nu V(n{:}N, d{:}B) = (d \implies V(n+1, d)) \wedge W(n, d) \big)$$
$$\big(\mu W(n{:}N, d{:}B) = d \vee (d \wedge W(n+1, d)) \big) \tag{2}$$

Currently, instantiation of the above equation systems leads to two infinite BESs. We will use Eqn. (2) as a running example in Section 4. □

3   Predicate Formula Normal Form

Manipulations and comparisons of formal objects typically benefit from the use (and existence) of a normal form for such objects. For this reason, we introduce a normal form for predicate formulae, which immediately implies a normal form for equation systems. Throughout this paper, we will then assume equation systems in normal form.

Definition 3. *A predicate formula is said to be in* Predicate Formula Normal Form *(PFNF) if it has the following form:*

$$Q_1 v_1{:}V_1. \cdots Q_n v_n{:}V_n.\ h \wedge \bigwedge_{i \in I} \big(g_i \implies \bigvee_{j \in J_i} X^j(e^j) \big)$$

where $X^j \in \mathcal{X}$, $Q_i \in \{\forall, \exists\}$, I is a (possibly empty) finite index set, each J_i is a non-empty finite index set, and h and every g_i are simple formulae.

Note that here J_i is used to index a set of occurrences of not necessarily different variables. For instance, $(n > 0 \implies X(3) \vee X(5) \vee Y(6))$ is a formula complying to the definition of PFNF. As long as it does not lead to confusion, we stick to the convention to drop the typing of the quantified variables v_i.

Proposition 1. *Every predicate formula can be rewritten to an equivalent predicate formula in PFNF.*

The proof is constructive, by means of a structural induction; this immediately provides the basis for a normalisation algorithm. We should remark that transforming the disjunction of two PFNF formulae into a PFNF formula leads to an undesirable blow-up of the formula size. However, when used, as here, in the context of equation systems, this blow-up can be reduced to a linear blow-up by introducing new equations.

A static invariance check. PBES invariants [13] are simple predicates characterising a closed set of attainable values of the data parameters in an equation system. They are very useful for simplifying and solving complex equation systems, as demonstrated in several case studies [13,8]. However, finding the right invariants is not easy, partly because the invariance condition quantifies over arbitrary predicate variable environments. Using PFNF, however, the invariance condition of [13] can be recast to one that can be checked statically. For completeness' sake, we first repeat the definition of an invariant.

Definition 4. *The simple function $f:V \to$ Pred is said to be a global invariant for an equation system $\mathcal{E}$ iff $\mathcal{X} \supseteq V \supseteq$ bnd$(\mathcal{E})$ and for each $(\sigma X(\boldsymbol{d}_X{:}\boldsymbol{D}_X) = \phi)$ occurring in $\mathcal{E}$, we have:*

$$f(X) \wedge \phi \;\leftrightarrow\; (f(X) \wedge \phi) \left[_{X_i \in V} (f(X_i) \wedge X_i(\boldsymbol{d}_{X_i}))_{\langle \boldsymbol{d}_{X_i} \rangle} / X_i \right].$$

Note that $\psi \left[_{X_i \in V} \phi(X_i)/X_i \right]$ stands for a simultaneous syntactic substitution of $\phi(X_i)$ for every X_i from V in ψ. The invariance condition basically states that the right-hand sides of all equations should be insensitive to strengthening all predicate variable occurrences with their corresponding simple formula. The following theorem provides an easy to check criterion implying the invariance condition.

Theorem 1. *Let $\mathcal{E}$ be an equation system where every equation k is in PFNF:*

$$\left(\sigma_k X_k(\boldsymbol{d}_{X_k}{:}\boldsymbol{D}_k) = \mathsf{Q}_1^k v_1 . \cdots \mathsf{Q}_{n_k}^k v_{n_k} . \, (h^k \wedge \bigwedge_{i \in I_k} (g_i^k \implies \bigvee_{j \in J_i} X^j(\boldsymbol{e}^j)))\right).$$

Then the simple function $f:V \to$ Pred is a global invariant for $\mathcal{E}$ if for each k:

$$\bigwedge_{i \in I_k} \bigwedge_{j \in J_i} \left((f(X_k) \wedge h^k \wedge g_i^k) \to f(X^j)[\boldsymbol{e}^j / \boldsymbol{d}_{X^j}] \right) \qquad (\iota_k)$$

$\square$

The proof is a tedious and semantically rather involved exercise leading to f satisfying Definition 4 under all data and predicate environments. It makes essential use of the fact that condition (ι_k) implicitly converts all quantifiers $\mathsf{Q}_1^k \ldots \mathsf{Q}_{n_k}^k$ into universal quantifiers. Note that (ι_k) is not a necessary condition for f to be an invariant, as demonstrated by the following example which makes use of the fact that all existential quantifiers are converted to universal quantifiers.

Example 2. Consider the equation system $\mathcal{E}$ given below:

$$\mathcal{E} :\equiv \left(\mu X(n{:}N) = \exists m{:}N.\, (m > 5 \implies Y(n))\right) \left(\nu Y(n{:}N) = Y(n+1)\right).$$

Let us define the simple function f as $f(X) = \top, f(Y) = \bot$. Condition (ι_k) in this case requires $\forall m{:}N.\, (\top \wedge m > 5 \implies \bot)$, which does not hold. Still, f is an invariant for $\mathcal{E}$, since it satisfies the invariant condition: $(\top \wedge \exists m{:}N.\, (m > 5 \implies Y(n))) \leftrightarrow (\top \wedge \exists m{:}N.\, (m > 5 \implies (\bot \wedge Y(n))))$. $\square$

It can be proven (not trivially) that the condition above is actually necessary in the case of quantifier-free equation systems, but we pose it as an interesting open problem whether condition (ι_k) can be modified (or some other condition can be thought up) to serve as a sufficient and necessary condition without employing a quantification over predicate environments.

4   Redundant and Constant Parameter Detection and Elimination

A part of the complexity of an equation system stems from the arity of the involved predicate variables and the types of these. Reducing an equation system's complexity can thus be achieved by minimising the complexity of the formal data parameters, either by removing them or by (implicitly) reducing their types. However, such operations are not always sound. In Sections 4.1 and 4.2, we develop algorithms that achieve these types of reductions without compromising soundness.

4.1   Parameter Elimination

The type of a predicate variable X is said to contain *redundancy* with respect to an environment if it relies on one or more values that do not manifest themselves in the truth of X using this environment.

Definition 5. *Given an environment η and a predicate variable X with signature $D_1 \times \cdots \times D_n \to B$, then a sort D_i ($1 \leq i \leq n$) is redundant with respect to η if for all values $v, w \in \mathbb{D}_i$, and $v_j \in \mathbb{D}_j$ for $j \neq i$, we have*

$$\eta(X)(v_1, \ldots, v, \ldots, v_n) = \eta(X)(v_1, \ldots, w, \ldots, v_n)$$

A semantic analysis of redundancy is neither feasible, nor desirable for most complex equation systems, so the best that can be achieved is to approximate the set of redundant sorts. We start by formalising the concept of influence.

Definition 6. *Let ρ be a predicate function in PFNF:*

$$\mathsf{Q}_1 v_1. \cdots \mathsf{Q}_n v_n.\, h \wedge \bigwedge_{i \in I} \left(g_i \implies \bigvee_{j \in J_i} X^j(e^j) \right)$$

We define the dependence set $\mathsf{dep}(\rho)$ and the significance set $\mathsf{sig}(\rho)$ as follows:

1. $\mathsf{dep}(\rho) =_{def} \bigcup_{i \in I} \{X^j(e^j) \mid j \in J_i\}$
2. $\mathsf{sig}(\rho) =_{def} \bigcup_{i \in I} \mathcal{FV}(\mathsf{Q}_1 v_1. \cdots \mathsf{Q}_n v_n.\, h \wedge g_i)$

The influence of a set of predicate functions on predicate variables is modelled by means of an influence graph. Recall that the i-th element of a vector d is denoted $d[i]$.

Definition 7. *Let $\mathcal{E} = (\sigma_1 X_1(d_{X_1}{:}D_{X_1}) = \phi_{X_1}) \cdots (\sigma_n X_n(d_{X_n}{:}D_{X_n}) = \phi_{X_n})$ be an equation system. The* marked influence graph $G(\mathcal{E}) = (V, \longrightarrow, M)$ *of $\mathcal{E}$ is a directed graph where:*

- $V = \{(X_i, j) \mid 1 \leq i \leq n \text{ and } 1 \leq j \leq \mathsf{arity}(D_{X_i})\}$;
- $\longrightarrow\, \subseteq V \times V$ *is the* transition relation, *defined by*

$$(X_i, k) \longrightarrow (X_j, l) \text{ iff } X_j(e) \in \mathsf{dep}(\phi_{X_i}) \text{ and } d_{X_i}[k] \in \mathcal{FV}(e[l])$$

- $M \subseteq V$ *is the* initial marking, *defined by*

$$M = \{(X_i, j) \mid 1 \leq i \leq n \text{ and } d_{X_i}[j] \in \mathsf{sig}(\phi_{X_i})\}$$

Intuitively, in a marked influence graph $G(\mathcal{E})$, the initial marking M is the set of variables that influences the truth of the simple formulae that occur in the predicate formulae of the equation system $\mathcal{E}$. The transition relation $\longrightarrow$ formalises the direct and indirect influence that formal parameters can have on the value of other formal parameters.

Example 3. Consider the equation system from Eqn. (2) (Example 1), first brought into PFNF. The marked influence graph is depicted on the right, where the marked states are black and non-marked states are white. $\square$
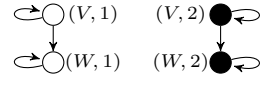
Next, we define the set of positive redundant variables as follows:

$$\mathcal{R} = \{ \boldsymbol{d}_{X_i}[j] \mid (X_i, j) \not\longrightarrow^* (X_k, l) \text{ and } (X_k, l) \in M \} \tag{3}$$

Computing the set $\mathcal{R}$ requires $\mathcal{O}(\mid \longrightarrow \mid)$ steps at most using, e.g., a standard least fixed point computation, depth-first or breadth-first search. We refrain from spelling out such an algorithm.

Definition 8. *Let* $\mathcal{E} = (\sigma_1 X_1(\boldsymbol{d}_{X_1}{:}\boldsymbol{D}_{X_1}) = \phi_{X_1}) \cdots (\sigma_n X_n(\boldsymbol{d}_{X_n}{:}\boldsymbol{D}_{X_n}) = \phi_{X_n})$, *where each* ϕ_{X_k} *is of the form:*

$$\mathsf{Q}_1^k v_1. \cdots \mathsf{Q}_{n_k}^k v_{n_k}. \, h^k \wedge \bigwedge_{i \in I^k} \left(g_i^k \implies \bigvee_{j \in J_i^k} X^j(\boldsymbol{e}^j) \right)$$

The reduction *of* $\mathcal{E}$*, denoted* $\widehat{\mathcal{E}}$ *is the equation system*

$$(\sigma_1 \widehat{X_1}(\widehat{\boldsymbol{d}_{X_1}}{:}\widehat{\boldsymbol{D}_{X_1}}) = \widehat{\phi_{X_1}}) \cdots (\sigma_n \widehat{X_n}(\widehat{\boldsymbol{d}_{X_n}}{:}\widehat{\boldsymbol{D}_{X_n}}) = \widehat{\phi_{X_n}})$$

where for every k *(*$1 \leq k \leq n$*), we define the following:*

1. $\widehat{\boldsymbol{d}_{X_k}}$ *is the vector* $\boldsymbol{d}_{X_k}$ *from which the parameters* $\boldsymbol{d}_{X_k}[i] \in \mathcal{R}$ *have been removed;*
2. $\widehat{\boldsymbol{D}_{X_k}}$ *is the vector* $\boldsymbol{D}_{X_k}$ *from which the types of* $\boldsymbol{d}_{X_k}[i]$ *have been removed;*
3. $\widehat{\phi_{X_k}} :\equiv \mathsf{Q}_1^k v_1. \cdots \mathsf{Q}_{n_k}^k v_{n_k}. \, h^k \wedge \bigwedge_{i \in I^k} \left(g_i^k \implies \bigvee_{j \in J_i^k} \widehat{X^j}(\widehat{\boldsymbol{e}^j}) \right)$*, where* $\widehat{\boldsymbol{e}^j}$ *is the vector* $\boldsymbol{e}^j$ *from which expressions* $\boldsymbol{e}^j[i]$ *with* $\boldsymbol{d}_{X^j}[i] \in \mathcal{R}$ *have been removed.*

The reduction of an equation system basically consists of a syntactic manipulation of predicate variable typings and predicate variable occurrences. It introduces a new set of predicate variables that are linked to the original predicate variables in the equation system. The typing of these newly introduced predicate variables is of lesser complexity than the typing of the original predicate variables. In particular, if the original equation system contains a predicate variable X_i of type $\mathbb{D}_{X_i}$, then the associated predicate variable $\widehat{X_i}$ is of type $\widehat{\mathbb{D}_{X_i}}$, which is based on $\widehat{\boldsymbol{D}_{X_i}}$. For elements $\boldsymbol{w}$ of type $\mathbb{D}_{X_i}$, we denote the corresponding reduced element by $\widehat{\boldsymbol{w}}$. We have the following two properties:

Lemma 1. *If* $\mathcal{E}$ *is data-closed, then so is* $\widehat{\mathcal{E}}$.

Proof. From the observation that $\widehat{\mathcal{E}}$ can only contain a free data variable if this is a formal parameter $\boldsymbol{d}_{X_k}[j]$ for some equation for $\widehat{X_k}$ that does not occur in the parameter list of this equation. This leads to a contradiction, based on the definition of $\mathcal{R}$ and $\widehat{\mathcal{E}}$. □

Lemma 2. *Let $\mathcal{E}$ be a data-closed equation system. Let $(\sigma_k X_k(\boldsymbol{d}_{X_k}:\boldsymbol{D}_{X_k}) = \phi_{X_k})$ be an arbitrary equation in $\mathcal{E}$. Let η be an environment for which $\eta(X)(\boldsymbol{v}) = \eta(\widehat{X})(\widehat{\boldsymbol{v}})$ for all $\boldsymbol{v}$ and $X \in \mathsf{occ}(\phi_{X_k})$. Then:*

$$\forall \varepsilon : \; [\![\phi_k]\!]\,\eta\varepsilon = [\![\widehat{\phi_k}]\!]\,\eta\varepsilon$$

Proof. Without loss of generality, ϕ_{X_k} is in PFNF. The proof then follows from a repeated application of the semantics. □

The following theorem demonstrates that the elimination of positively redundant formal parameters from an equation system does not affect the solution of the equation system.

Theorem 2. *Let $\mathcal{E}$ be a data-closed equation system. Let η, ε be arbitrary environments. If for all $X \in \mathsf{free}(\mathcal{E})$ we have $\eta(X)(\boldsymbol{v}) = \eta(\widehat{X})(\widehat{\boldsymbol{v}})$ for all $\boldsymbol{v}$, then for all $X_k \in \mathsf{bnd}(\mathcal{E})$ and all $\boldsymbol{v}$:*

$$[\![\mathcal{E}]\!]\,\eta\varepsilon(X_k(\boldsymbol{v})) = [\![\widehat{\mathcal{E}}]\!]\,\eta\varepsilon(\widehat{X_k}(\widehat{\boldsymbol{v}}))$$

Proof. By means of an induction on $|\mathcal{E}|$. The base case follows immediately. The induction requires a case distinction, a transfinite approximation and Lemmas 1 and 2. □

Corollary 1. *In case $\mathcal{E}$ is data-closed and closed, we obtain the following result:*

$$\forall \eta, \varepsilon : \; \forall X_l \in \mathsf{bnd}(\mathcal{E}) : \; [\![\mathcal{E}]\!]\,\eta\varepsilon(X_l(\boldsymbol{v})) = [\![\widehat{\mathcal{E}}]\!]\,\eta\varepsilon(\widehat{X_l}(\widehat{\boldsymbol{v}})))$$

Example 3. Consider the equation system from Eqn. (2) from Example 1, brought into PFNF, and name it $\mathcal{E}$. From its marked influence graph, we find $\mathcal{R} = \{(V, 2), (W, 2)\}$. This means that equation system (2) can be reduced to the equation system $\widehat{\mathcal{E}}$, where:

$$\widehat{\mathcal{E}} :\equiv \left(\nu\widehat{V}(d:B) = (d \implies \widehat{V}(d)) \wedge \widehat{W}(d)\right) \left(\mu\widehat{W}(d:B) = d \wedge (\neg d \implies \widehat{W}(d))\right)$$

We find that for all $j \in \mathbb{N}$ and $b \in \mathbb{B}$, we have $[\![\mathcal{E}]\!]\,\eta\varepsilon(X(j,b)) = [\![\widehat{\mathcal{E}}]\!]\,\eta\varepsilon(\widehat{X}(b))$ for $X \in \{V, W\}$ and all η, ε. Moreover, a full instantiation of $\widehat{\mathcal{E}}$ (see [4]) leads to a BES consisting of four equations:

$$(\nu\widehat{V}_\top = \widehat{V}_\top \wedge \widehat{W}_\top) \; (\nu\widehat{V}_\bot = \widehat{W}_\bot) \; (\mu\widehat{W}_\top = \top) \; (\mu\widehat{W}_\bot = \bot)$$

where variable $\widehat{X}_d$ encodes $\widehat{X}(d)$ for $d \in \mathbb{B}$ and $X = V, W$. The above BES immediately leads to the answer *true* for variables $\widehat{V}_\top, \widehat{W}_\top$ and *false* for the variables $\widehat{V}_\bot, \widehat{W}_\bot$. Hence, using redundant parameter elimination, instantiation allows for solving equation systems that could not be solved before using this technique, solving verification problems that we could not solve before. □

4.2   Detection of Constants

As in the previous section, let $\mathcal{E}$ be an equation system, where every equation l ($1 \leq l \leq N$) is in PFNF, and let κ be a *target predicate formula* (i.e., a formula whose truth we wish to assess in the context of $\mathcal{E}$), without any free data variables, in PFNF. In this section, we develop an algorithm that automatically computes an invariant that associates constants to the formal parameters of predicate variables. To this end, we will be using a special type of simple functions called *ground functions*.

Definition 9. *A predicate formula $p \in$ Pred is a ground predicate for a variable X of an equation system $\mathcal{E}$ if $p \equiv \bot$ or $p \equiv (\boldsymbol{d}_X = \boldsymbol{c})$, where $\boldsymbol{c}$ is a partially instantiated list, i.e. for all indices j, $\boldsymbol{c}[j] \in \boldsymbol{D}_X[j] \cup \{\boldsymbol{d}_X[j]\}$. A simple function $g{:}\mathsf{occ}(\mathcal{E}) \to$ Pred is a ground function if, for all variables X, $g(X)$ is a ground predicate.*

Here $\boldsymbol{d}_X = \boldsymbol{c}$ is a shortcut for $\bigwedge_{1 \leq i \leq \mathsf{arity}(X)}(\boldsymbol{d}_X[i] = \boldsymbol{c}[i])$. Ground predicates formalise assertions about the values associated to the data parameters. To each parameter i, either a constant from its value domain, or its name $\boldsymbol{d}_X[i]$ is associated. In the latter case, the corresponding assertion is thus $\boldsymbol{d}_X[i] = \boldsymbol{d}_X[i]$, evaluating to $\top$.

The set of ground predicates for X in $\mathcal{E}$ is $\mathsf{GPred}_{\mathcal{E},X}$ and the set of ground functions for $\mathcal{E}$ is $\mathsf{GFunc}_{\mathcal{E}}$. The binary operator SUP takes two ground predicates of the same variable and yields the supremum of these ground predicates; it remains undefined for ground predicates of different variables.

$$\mathsf{SUP}((\boldsymbol{d}_X = \boldsymbol{c}), (\boldsymbol{d}_X = \boldsymbol{c}')) =_{def} (\boldsymbol{d}_X = \boldsymbol{c}'')), \text{ where for all } 1 \leq i \leq n:$$
$$\boldsymbol{c}''[i] = \begin{cases} \boldsymbol{c}[i], & \text{if } \boldsymbol{c}[i] = \boldsymbol{c}'[i] \\ \boldsymbol{d}_X[i], & \text{otherwise} \end{cases} \tag{4}$$

For instance, if X has arity 3, then $\mathsf{SUP}((\boldsymbol{d}_X = \langle 2, 0, \boldsymbol{d}_X[3]\rangle), (\boldsymbol{d}_X = \langle 2, 5, \boldsymbol{d}_X[3]\rangle))$ yields $(\boldsymbol{d}_X = \langle 2, \boldsymbol{d}_X[2], \boldsymbol{d}_X[3]\rangle)$. The operator SUP extends naturally to sets of ground predicates of the same variable; as a convention, we set $\mathsf{SUP}(\emptyset) =_{def} \bot$.

We call $X(\boldsymbol{e})$, with $\boldsymbol{e}$ a list of data expressions, an *instantiated occurrence* of X and we denote the set of all instantiated occurrences of predicate variables in ϕ by $iocc(\phi)$. From any such occurrence $X(\boldsymbol{e})$, we can extract a ground predicate by retaining only those data expressions which are constants:

$$\mathtt{gpred}(X(\boldsymbol{e})) =_{def} (\boldsymbol{d}_X = \boldsymbol{c}), \text{ with } \boldsymbol{c}[i] = \begin{cases} c, & \text{if } \boldsymbol{e}[i] \leftrightarrow c \text{ and } c \in \boldsymbol{D}_X[i] \\ \boldsymbol{d}_X[i], & \text{otherwise.} \end{cases} \tag{5}$$

Finally, let ϕ be a formula in PFNF: $Q_1 v_1 \ldots Q_n v_n.h \land \bigwedge_{i \in I} g_i \implies \bigvee_{j \in J_i} X^j(\boldsymbol{e}^j)$. Then the *guard* of an instantiation $X^j(\boldsymbol{e}^j)$ in ϕ, written $\mathtt{guard}(X^j(\boldsymbol{e}^j), \phi)$ is defined as $h \land g_i$.

A constant detection algorithm. The recursive function ga generates successive ground functions that approximate the parameter lists of $\mathcal{E}$'s predicate variables reached when starting instantiation of $\mathcal{E}$ from target predicate formula κ.

$\text{ConstElm}(\mathcal{E}, \kappa))$

for $X \in \text{occ}(\mathcal{E})$,
$$\text{ga}_0(X) \coloneqq \text{SUP}(\{\text{gpred}(X(e))| \ X(e) \in iocc(\kappa) \wedge \text{guard}(X(e), \kappa) \not\rightarrow \bot\})$$
for $X \in \text{occ}(\mathcal{E})$,
$$\text{ga}_{n+1}(X) \coloneqq \text{SUP}(\{\text{ga}_n(X)\} \cup \bigcup_{\text{ga}_n(Y) \equiv (d_Y = e)} \{\text{gpred}(X(e'))|$$
$$X(e') \in iocc(\phi_Y[e/d_Y]) \wedge \text{guard}(X(e'), \phi_Y[e/d_Y]) \not\rightarrow \bot\})$$
$$\text{Output}: \quad \text{gi} \equiv \bigvee_{n \geq 0} \text{ga}_n$$

Both in the definition of GPred and in the algorithm, the existence of a sound decision method for $\phi \leftrightarrow \psi$ is assumed. Usually a very simple one, like syntactic equivalence, will produce meaningful enough invariants.

Correctness. We next give a formal argument for the correctness of the constant detection algorithm; that is, we show that the output of the algorithm yields an invariant of the original PBES which preserves the truth of κ.

The function ga_n captures the information gathered from the arguments of the predicate variables after n substitution steps. $\text{ga}_n(X_k) = \bot$ has the intuitive meaning that X_k is unreachable from κ within n substitution steps and $\text{ga}_n(X_k) = \top$ holds when none of X_k's arguments remain constant. We have the following simple observation:

Lemma 3. *For all $n \geq 0$ and all $X \in \text{occ}(\mathcal{E})$, $\text{ga}_n(X) \rightarrow \text{ga}_{n+1}(X)$.*

Proof (sketch). It is easy to prove that $p \rightarrow \text{SUP}(S)$ for all $p \in S$ with S a set of ground predicates for X. Then, from the way ga_{n+1} is constructed, namely $\text{ga}_{n+1}(X) = \text{SUP}(\{\text{ga}_n(X)\} \cup S)$, we immediately conclude that $\text{ga}_n(X) \rightarrow \text{ga}_{n+1}(X)$. □

Lemma 4. $\text{ConstElm}(\mathcal{E}, \kappa)$ *terminates for every $\mathcal{E}$ and κ.*

Proof (sketch). In the $\text{ConstElm}(\mathcal{E}, \kappa)$ computation, an index N is reached for which $\text{ga}_N \equiv \text{ga}_{N+1}$ (for every predicate variable X, $\text{ga}_N(X) \equiv \text{ga}_{N+1}(X)$); for this N, we have $\text{ga}_N = \text{ga}_{N+k}$ for every k (i.e., ga is stable). The first part of this claim follows from the finiteness of $\text{bnd}(\mathcal{E})$ and the observation that there is a decreasing measure that can be associated to the $\text{ga}_0 \ldots \text{ga}_n \ldots$ sequence (viz., the number of constants occurring in ga_n). The second part follows by induction. The output of the algorithm is then $\bigvee_{0 \leq i \leq N} \text{ga}_N$; following Lemma 3, this is equivalent to ga_N. □

The theorem below states that the algorithm indeed yields valid invariants.

Theorem 3. *The output of the algorithm $\text{ConstElm}(\mathcal{E}, \kappa)$ is a global invariant for $\mathcal{E}$.*

Proof (sketch). Lemma 4 shows that gi is in fact ga_N for some sufficiently large N. We prove by contradiction that ga_N satisfies the sufficient condition from Theorem 1. The argument essentially uses Lemma 3. Consequently, ga_N is a global invariant. □

Using the invariant gi computed by ConstElm, we can now strengthen the original PBES. The strengthened system is, according to the definition from [13]:

$$\text{red}(\epsilon) = \epsilon$$
$$\text{red}((\sigma X(d_X{:}D_X) = \phi) \, \mathcal{E}') = (\sigma X(d_X{:}D_X) = \text{gi}(X) \wedge \phi) \, \text{red}(\mathcal{E}')$$

Note that if $\mathrm{gi}(X) \equiv (\boldsymbol{d}_X = \boldsymbol{c})$ then $\mathrm{gi}(X) \wedge \phi$ is in fact logically equivalent to $\phi[\boldsymbol{c}/\boldsymbol{d}_X]$, meaning that the number of free variables in ϕ decreases. Using the redundant parameter elimination technique of the previous section, the equation system can then be reduced. So, red indeed reduces the complexity of equation systems. It suffices to show that this strengthening preserves the truth for κ:

Theorem 4. *Let η and ϵ be arbitrary predicate and data environments, respectively. Then $[\![\kappa]\!]([\![\mathcal{E}]\!]\eta\epsilon)\epsilon = [\![\kappa]\!]([\![\mathrm{red}(\mathcal{E})]\!]\eta\epsilon)\epsilon.$* □

Proof (sketch). The preservation of κ's solution follows from Corollary 2 of [13] and the identity $\kappa \leftrightarrow \kappa \left[_{X_i \in V} (\mathrm{gi}(X_i) \wedge X_i(d_{X_i}))_{\langle d_{X_i} \rangle} / X_i \right]$, which can be shown by a series of calculations. □

Example 4. Let κ be the target formula $\forall v{:}\mathbb{N}.\ X(1, v)$, and $\mathcal{E}$ defined as:

$$(\mu X(m{:}\mathbb{N}, n{:}\mathbb{N}) = m \leq 10 \Rightarrow (X(m, n+1) \vee Y(m)))$$
$$(\nu Y(p{:}\mathbb{N}) \quad = X(p, 0) \wedge (p \geq 5 \Rightarrow Z(p)))$$
$$(\mu Z(q{:}\mathbb{N}) \quad = q \leq 10)$$

The `ConstElm` algorithm produces the following approximations:

$$
\begin{array}{lll}
\mathrm{ga}_0(X) = (m = 1 \wedge n = n) & \mathrm{ga}_0(Y) = \bot & \mathrm{ga}_0(Z) = \bot \\
\mathrm{ga}_1(X) = (m = 1 \wedge n = n) & \mathrm{ga}_1(Y) = (p = 1) & \mathrm{ga}_1(Z) = \bot \\
\mathrm{ga}_2(X) = (m = 1 \wedge n = n) & \mathrm{ga}_2(Y) = (p = 1) & \mathrm{ga}_2(Z) = \bot
\end{array}
$$

Strengthening each equation with the invariant gi (which assigns $m = 1$ to X, $p = 1$ to Y and $\bot$ to Z), and simplifying the resulting right-hand sides of the equations, we obtain the following reduced equation system:

$$(\mu X(m{:}\mathbb{N}, n{:}\mathbb{N}) = X(1, n+1) \vee Y(1))$$
$$(\nu Y(p{:}\mathbb{N}) \quad = X(1, 0))$$
$$(\mu Z(q{:}\mathbb{N}) \quad = \bot)$$

Using the technique from the previous section, we find that formal parameters m, n, p and q all become redundant. The solution to this system of equations has $\bot$ as a solution for X, Y and Z, which leads to the solution $\bot$ for κ. Note that the instantiation solving technique does not terminate on the original equation system $\mathcal{E}$, nor does the redundant parameter elimination technique remove formal parameters from $\mathcal{E}$. □

Complexity and implementation. Denote the number of predicate variables in $\mathrm{occ}(\mathcal{E})$ by v, the maximum length for the argument list of a predicate variable by l, and the maximum number of occurrences of one variable in all the right-hand sides of $\mathcal{E}$'s equations by o. For each iteration n, ga_n can be computed in $\mathcal{O}(v \times l \times o)$ (ignoring the complexity of rewriting that may be necessary), since there are v variables and for each $\mathrm{ga}_n(X)$, the SUP of a set of at most o ground predicates is computed. This requires comparisons of arrays of length l. Every variable $\boldsymbol{d}_X[i]$ from an argument list can appear in the ground approximations of X as a constant ($\boldsymbol{d}_X[i] = c \in \boldsymbol{D}_X[i]$) or as variable $\boldsymbol{d}_X[i] = \boldsymbol{d}_X[i]$. Once its assertion becomes $\boldsymbol{d}_X[i] = \boldsymbol{d}_X[i]$, it will never become of the constant type

again. Since the total number of constant assertions is decreasing with every iteration (see also the proof of Lemma 3), and since there are at most $v \times l$ data variables in the system, the maximum number of iterations until stabilisation is $v \times l$. Hence, an upper bound on the total cost of `ConstElm` is $\mathcal{O}(v^2 \times l^2 \times o)$.

In practice, checking whether the guards are unsatisfiable requires careful bookkeeping and can be inefficient. A sound solution is to over-approximate all guards to $\top$, leading to a much quicker algorithm (which may compute weaker invariants), while preserving soundness (the same proof applies).

5   Experiments

For conducting our experiments, we have used the tool-suite mCRL2[1], which implements techniques from [7,4] and for which we implemented the redundant parameter elimination and the constant parameter elimination methods. All experiments described in the remainder of this section have been conducted on a Dual Core, 2.6GHz AMD Opteron Processor running Linux with 128Gb memory.

Redundant Parameter Elimination. The first series of experiments consists of an analysis of several communication protocols from the literature: the Alternating Bit Protocol, the Positive Acknowledgement Retransmission Protocol, the Concurrent Alternating Bit Protocol and variations of the Sliding Window Protocol with different buffer size. Note that none of these system descriptions could be simplified using the related parameter elimination techniques from [6] without manually changing the descriptions.

The verification problem that we encoded for each of these protocols is deadlock-freedom. We varied the size of the set of messages M that can be communicated from $2, 4, 8$ to ∞, and measured the increase in the size of the BES that is obtained by instantiating the respective PBES before and after applying the redundant parameter elimination technique of Section 4.1, see Table 1. Running the latter algorithm takes less than .1 seconds for every equation system (the number of data parameters for each equation system varies from 13 to 22 per equation). Clearly, the size of the set of message has a significant impact on the size of the BESs, as witnessed e.g., for the SWP with buffer size 4: instantiating the equation system for the SWP with $|M| = 4$ is not viable in a reasonable amount of time. In contrast, the redundant parameter elimination method detects that the content of the messages is irrelevant for deadlock-freedom and therefore eliminates all references to messages from the equation system, resulting in small BESs. A similar reduction in size can be obtained by first abstracting out the data from the protocol descriptions and then checking the resulting systems for deadlock-freedom; this, however, requires in-depth knowledge about the behaviours of the systems.

A second batch of experiments conducted using the same protocols and the same setup is to verify whether a sender can infinitely often send a particular (constant) message m, which is formalised by the following formula of fixed point alternation depth 2: $\nu X. \mu Y. \langle r(m) \rangle X \vee \langle \neg r(m) \rangle Y$, where the action r models the sending of the message to the communications protocol (which is $\underline{r}$eceiving the message). The number of data parameters for the equation systems again varies from 13 to 22 per equation). Unlike for

---

[1] `http://www.mcrl2.org`

Table 1. The effect of redundant parameter elimination in equation systems encoding (1) deadlock-freedom ($\nu X.\ [\top]X \wedge \langle\top\rangle\top$) and (2) infinitely-often sending a constant message ($\nu X.\mu Y.\ \langle r(m)\rangle X \vee \langle\neg r(m)\rangle Y$) of various communication protocols using a set M of messages. Here, the x in x/y stands for the size of the BES before elimination of redundancy and the y in x/y stands for the size of the BES after elimination of redundancy.

Property (1)

$|M| \rightarrow$ Protocol $\downarrow$	2	4	8	∞
ABP	74 / 38	146 / 38	290 / 38	∞ / 38
PAR	91 / 47	179 / 47	355 / 47	∞ / 47
CABP	464 / 224	1,040 / 224	2,576 / 224	∞ / 224
One-bit SWP	324 / 144	900 / 144	2,916 / 144	∞ / 144
SWP (buffer size 2)	14,064 / 1,860	140,352 / 1,860	$1.7 * 10^6$ / 1,860	∞ / 1,860
SWP (buffer size 4)	$2.6 * 10^6$ / 43,320	nc / 43,320	nc / 43,320	∞ / 43,320

Property (2)

$|M| \rightarrow$ Protocol $\downarrow$	2	4	8	∞
ABP	77 / 41	149 / 41	293 / 41	∞ / 41
PAR	95 / 51	183 / 51	359 / 51	∞ / 51
CABP	513 / 257	1,121 / 257	2,721 / 257	∞ / 257
One-bit SWP	379 / 181	991 / 181	3,079 / 181	∞ / 181
SWP (buffer size 2)	17,809 / 2,545	163,393 / 2,545	$1.9 * 10^6$ / 2,545	∞ / 2,545
SWP (buffer size 4)	$3.5 * 10^6$ / 64,609	nc / 64,609	nc / 64,609	∞ / 64,609

the deadlock freedom property, the presence of the data message in the system description is vital, which means that abstracting the data away is no option. First applying the redundant parameter elimination technique from [6] therefore yields no improvement. The results of our experiments are depicted in the second matrix in Table 1 and show a similar trend as the first batch of experiments. It is important to note that the resulting BESs are also alternating. Thus, size really becomes a problem as the best, known algorithms for solving BESs are exponential in the fixed point alternation depth and have as base the size of the BES. Large BESs thus quickly become intractable.

Constant Parameter Elimination. As demonstrated in [13,8], well chosen invariants help to simplify equation systems, so that rewriting becomes less of a bottleneck in instantiating. Figure 1 clearly illustrates this effect: applying a constant parameter elimination
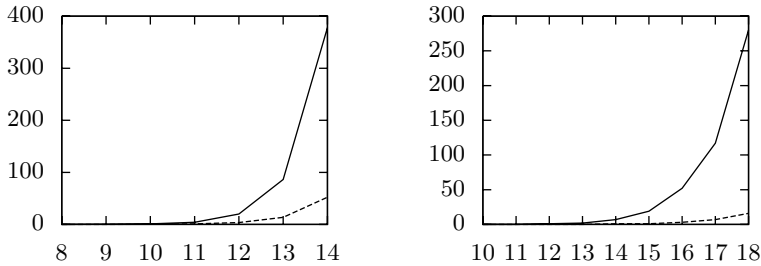

Fig. 1. The effect of constant parameter elimination on the instantiation time required to obtain a BES from an equation system encoding the deadlock-freedom property for n dining philosophers (left) and for n Milner schedulers (right). Time is measured in minutes (y-axis).

in the equation systems that encode deadlock-freedom in n dining philosophers and a token ring of n Milner schedulers, respectively. The time required to solve the original equation systems increases exponentially with increasing n in both cases (continuous lines). In contrast, after applying a constant parameter elimination, the increase in time to compute the resulting equation system is modest (dashed lines). Removing the constant parameters requires little time: < 10 seconds for the most complex case.

6   Summary

We have devised algorithms that reduce the complexity of PBESs; this is achieved by detecting and removing parameters that do not contribute to the solution of a PBES and by removing constant parameters. Our experiments show that the algorithms are very effective at reducing the size of the BESs that can be computed from the PBESs. This means that the complexity of solving the PBES via a resolution of the BES can be reduced as well, which is particularly important for alternating BESs.

As we observed and proved, a sufficient invariance condition for PBESs can be stated that does not require a quantification over arbitrary predicate environments. Our constant detection algorithm is a special case of an invariance detection algorithm for PBESs, based on the above-mentioned observation; it is therefore interesting future work to extend the constant detecting algorithm in order to detect more complex classes of invariants.

References

1. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized boolean equation systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 120–135. Springer, Heidelberg (2007)
2. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
4. van Dam, A., Ploeger, B., Willemse, T.A.C.: Instantiation for parameterised boolean equation systems. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 440–454. Springer, Heidelberg (2008)
5. Gallardo, M.M., Joubert, C., Merino, P.: Implementing influence analysis using parameterised boolean equation systems. In: Proc. ISOLA 2006. IEEE Comp. Soc. Press, Los Alamitos (2006)
6. Groote, J.F., Lisser, B.: Computer assisted manipulation of algebraic process specifications. SIGPLAN Notices 37(12), 98–107 (2002)
7. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Sci. Comput. Program 56(3), 251–273 (2005)
8. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theor. Comput. Sci. 343(3), 332–369 (2005)
9. Hentze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: Proc. PLDI 1997. ACM, New York (1997)
10. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal transition systems: A foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, p. 155. Springer, Heidelberg (2001)

11. Mateescu, R.: Local model-checking of an alternation-free value-based modal mu-calculus. In: Proc. 2nd Int'l Workshop on VMCAI (September 1998)
12. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
13. Orzan, S., Willemse, T.A.C.: Invariants for parameterised boolean equation systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 187–202. Springer, Heidelberg (2008)
14. van de Pol, J.C., Valero Espada, M.: Modal abstractions in μCRL*. In: Proc. AMAST (2004)
15. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Mathematics 5(2), 285–309 (1955)
16. Watanabe, H., Nishizawa, K., Takaki, O.: A coalgebraic representation of reduction by cone of influence. In: Proc. of Workshop on Coalgebraic Methods in Computer Science, vol. 164(1), pp. 177–194 (2006)

Parametric Trace Slicing and Monitoring[*]

Feng Chen and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
{fengchen,grosu}@cs.uiuc.edu

Abstract. Analysis of execution traces plays a fundamental role in many program analysis approaches. Execution traces are frequently parametric, i.e., they contain events with parameter bindings. Each parametric trace usually consists of many *trace slices* merged together, each slice corresponding to a parameter binding. Several techniques have been proposed to analyze parametric traces, but they have limitations: some in the specification formalism, others in the type of traces they support; moreover, they share common notions, intuitions, even techniques and algorithms, suggesting that a fundamental understanding of parametric trace analysis is needed. This foundational paper gives the first solution to parametric trace analysis that is unrestricted by the type of parametric properties or traces that can be analyzed. First, a general purpose parametric trace slicing technique is discussed, which takes each event in the parametric trace and distributes it to its corresponding trace slices. This parametric trace slicing technique can be used in combination with any conventional, non-parametric trace analysis, by applying the latter on each trace slice. An online monitoring technique is then presented based on the slicing technique, providing a logic-independent solution to runtime verification of parametric properties. The presented monitoring technique has been implemented and extensively evaluated. The results confirm that the generality of the discussed techniques does not come at a performance expense when compared with existing monitoring systems.

1 Introduction and Motivation

Parametric traces, i.e., traces containing events with parameter bindings, abound in programming language executions, because they naturally appear whenever abstract parameters (e.g., variable names) are bound to concrete data (e.g., heap objects) at runtime. For example, if one is interested in analyzing collections and iterators in Java, then execution traces of interest may contain events createIter$\langle c\,i \rangle$ (iterator i is created for collection c), updateColl$\langle c \rangle$ (c is modified), and "next$\langle i \rangle$ (i is accessed using its next element method), instantiated for particular collection and iterator instances. Most properties of parametric traces are also parametric, i.e., refer to each particular parameter instance; for example, a property may be "collections are not allowed to change while accessed through

iterators", which is parametric in a collection and an iterator. To distinguish properties parametric in a set of parameters X from ordinary, non-parametric properties, we write them $\Lambda X.P$; for example, violations of the above parametric property expressed as a regular expression (here matches mean violations) can be "$\Lambda c, i.$ createIter$\langle c\, i \rangle$ next$\langle i \rangle^*$ updateColl$\langle c \rangle^+$ next$\langle i \rangle$". From here on we omit the event parameters in parametric properties when they are redundant; for example, we write "$\Lambda c, i.$ createIter next* updateColl$^+$ next" for the above.

Parametric properties, unfortunately, are very hard to formally verify and validate against real systems, mainly because of their dynamic nature and potentially huge or even unlimited number of parameter bindings. Let us extend the above example: in Java, one may create a collection from a map and use the collection's iterator to operate on the map's elements. A similar safety property is: "maps are not allowed to change while accessed indirectly through iterators". Its violation pattern is: "$\Lambda m, c, i.$ createColl (updateMap | updateColl)* createIter next* (updateMap | updateColl)$^+$ next", with two new parametric events createColl$\langle m, c \rangle$ (collection c is created from map m) and updateMap$\langle m \rangle$ (m is updated). All the events used in this property provide only partial parameter bindings (createColl binds only m and c, etc.), and parameter bindings carried by different events may be combined into larger bindings during the analysis; e.g., createColl$\langle m_1, c_1 \rangle$ can be combined with createIter$\langle c_1, i_1 \rangle$ into a full binding $\langle m_1, c_1, i_1 \rangle$, and also with createIter$\langle c_1, i_2 \rangle$ into $\langle m_1, c_1, i_2 \rangle$. It is highly challenging for a trace analysis technique to correctly and efficiently maintain, locate and combine trace slices for different parameter bindings, especially when the trace is long and the number of parameter bindings is large.

Parametric properties have been receiving growing interest in *runtime verification* (RV), as shown by the increasing number of RV systems supporting them, e.g., [3,1,14,12,7,13,15,9,5,4]. Most of these techniques tightly couple the handling of parameter bindings with the property checking, yielding monolithic but supposedly efficient monitors. For example, Tracematches [1] extends state machines with parameter bindings in order to support parametric regular pattern properties; a series of optimizations to the resulting data-structures make Tracematches one of the most efficient RV systems [3]. The major challenges these "monolithic monitor" approaches face are how to keep track of the status for each particular parameter instance during property checking, and how to correctly garbage-collect portions of the monitor as they become irrelevant [14,5,4]. Such couplings of parameter binding and property checking result in rather complex and property-formalism-specific algorithms, hard or impossible to adapt to other formalisms. For example, [3] builds upon a *finite* state machine skeleton associated to the underlying pattern, so it cannot be adapted to, e.g., context-free patterns; [14] stacks automata in order to support parametric context-free patterns, making it slower than Tracematches [3] and insensitive to certain events of interest (such as ends of procedures [15]); Eagle [4] has no garbage-collection due to its generality, causing prohibitive overhead [3].

JavaMOP [9] proposes a different solution, based on a complete decoupling of parameter binding from property checking. This separation allows the use

of "off-the-shelf" algorithms and techniques for non-parametric properties as plug-ins; e.g., JavaMOP supports several property specification formalisms, including regular expressions, temporal logics, and context-free patterns [15,9], all parametric. However, the technique currently supported by JavaMOP can only handle a limited type of traces, namely ones in which the first event for a particular property instance binds all the property parameters. This limitation prevents JavaMOP from supporting many useful parametric properties [3]. In this paper we show that the decoupling of parameter binding and property checking is not only possible without any limitation, but also very practical.

In spite of all the recent advances in parametric property and trace analysis, the following questions are still left largely open in their full generality: Given a parametric trace τ and a parametric property $\Lambda X.P$, what does it mean for τ to be a good or a bad trace for $\Lambda X.P$? How can we leverage, to the parametric case, knowledge and techniques to analyze conventional, non-parametric traces against conventional, non-parametric properties? In this paper we first formulate and then rigorously answer these questions and empirically validate our answer.

Contributions. Besides proposing a formal semantics to parametric traces, properties, and monitoring, we make two theoretical contributions and discuss an implementation that validates them empirically. Our first result is a general-purpose online parametric trace slicing algorithm (algorithm $\mathbb{A}\langle X\rangle$), which positively answers the question: given a parametric execution trace τ, can one effectively find the slices $\tau \restriction_\theta$ corresponding to each parameter instance θ without having to traverse the trace for each θ? Our second result, which builds upon the slicing algorithm, is an online monitoring technique (algorithms $\mathbb{B}\langle X\rangle$ and $\mathbb{C}\langle X\rangle$) for parametric properties, which separates handling of parameters from checking trace slices against the specified property. It positively answers the question: is it possible to monitor arbitrary parametric properties $\Lambda X.P$ against parametric execution traces τ, provided that the root non-parametric property P is monitorable using conventional monitors? Finally, we implemented and evaluated the proposed techniques and show empirically that their generality does not come at a performance expense when compared with existing monitoring systems.

Paper structure. Section 2 formalizes parametric events, traces and properties, defines trace slicing and discusses an online trace slicing algorithm. Section 3 presents our main techniques for parametric trace monitoring. Section 4 discusses implementation optimizations to the proposed monitoring technique and its evaluation. Section 5 summarizes related researches and Section 6 concludes.

Note. Proofs to all the claimed results can be found in [16].

2   Parametric Trace Slicing for Monitoring

In this section, we first define some basic notions (event, trace and property) and then present an online parametric trace slicing algorithm that provides the foundation for the online monitoring technique discussed in Section 3.

2.1   Events, Traces and Properties

Here we introduce the notions of event, trace and property, first non-parametric and then parametric. Trace slicing is then defined as a reduct operation that forgets all the events unrelated to the given parameter instance.

Definition 1. *Let $\mathcal{E}$ be a set of (non-parametric) events, called* **base events** *or simply* **events***. An $\mathcal{E}$-trace, or simply a (non-parametric)* **trace** *when $\mathcal{E}$ is understood or not important, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.*

Example. *(part 1 of simple running example)* Consider a certain resource (e.g., a synchronization object) that can be acquired and released during the lifetime of a given procedure (between its begin and end). Then $\mathcal{E}=\{\mathsf{acquire}, \mathsf{release}, \mathsf{begin}, \mathsf{end}\}$ and execution traces corresponding to this resource are sequences of the form "begin acquire acquire release end begin end", "begin acquire acquire", etc. For now there are no "good" or "bad" execution traces. □

Definition 2. *An $\mathcal{E}$-property P, or simply a (base or non-parametric)* **property***, is a function $P : \mathcal{E}^* \to \mathcal{C}$ partitioning the set of traces into categories $\mathcal{C}$. It is common, but not enforced, that $\mathcal{C}$ includes "validating", "violating", and "don't know" (or "?") categories. In general, $\mathcal{C}$, the co-domain of P, can be any set.*

Example. *(part 2)* Consider a regular expression specification, $(\mathsf{begin}(\epsilon \mid (\mathsf{acquire}$ $(\mathsf{acquire} \mid \mathsf{release})^* \mathsf{release}))\mathsf{end})^*$, stating that the procedure can (non-recursively) take place multiple times and, if the resource is acquired during the procedure then it is released by the end of the procedure. The validating traces are those matching the pattern, e.g., "begin acquire acquire release end begin end". At first sight, one may say that all the other traces are violating traces, because they are not in the language of the regular expression. However, there are two interesting types of such "violating" traces: ones which may still lead to a validating trace provided the right events will be received in the future, e.g., "begin acquire acquire", and ones which have no chance of becoming a validating trace, e.g. "begin acquire release acquire end". Therefore, we can pick $\mathcal{C}$ to be the set $\{\mathsf{validating}, \mathsf{violating}, \mathsf{don't\ know}\}$ and, for a given regular expression E, define its associated property $P_E : \mathcal{E}^* \to \mathcal{C}$ as follows: $P_E(w) = \mathsf{validating}$ iff w is in the language of E, $P_E(w) = \mathsf{violating}$ iff there is no $w' \in \mathcal{E}^*$ such that $w\,w'$ is in the language of E, and $P_E(w) = \mathsf{don't\ know}$ otherwise; this is the monitoring semantics of regular expressions in JavaMOP [9]. Other semantic choices are possible; for example, one may choose $\mathcal{C}$ to be the set $\{\mathsf{matching}, \mathsf{don't\ care}\}$ and define $P_E(w) = \mathsf{matching}$ iff w is in the language of E, and $P_E(w) = \mathsf{don't\ care}$ otherwise; this is the semantics of regular expressions in Tracematches [1]. □

We next extend the above definitions to the parametric case, i.e., events carrying concrete data instantiating abstract parameters.

Example. *(part 3)* Events acquire and release are parametric in the resource; if r is the name of the generic "resource" parameter and r_1 and r_2 are two concrete resources, then parametric acquire/release events have the form $\mathsf{acquire}\langle r \mapsto r_1\rangle$,

release$\langle r \mapsto r_2 \rangle$, etc. Not all events need carry instances for all parameters; e.g., the begin/end parametric events have the form begin$\langle \bot \rangle$ and end$\langle \bot \rangle$, where $\bot$, the partial map undefined everywhere, instantiates no parameter.   □

Let $[A \to B]/[A \rightharpoonup B]$ be the sets of total/partial functions from A to B.

Definition 3. (Parametric events and traces). *Let X be a set of **parameters** and let V be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events like in Def. 1, then let $\mathcal{E}\langle X \rangle$ be the set of corresponding **parametric events** $e\langle \theta \rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \rightharpoonup V]$. A **parametric trace** is a trace with events in $\mathcal{E}\langle X \rangle$, that is, a word in $\mathcal{E}\langle X \rangle^*$.*

To simplify writing, we occasionally assume the parameter values set V implicit.

Example. *(part 4)* A parametric trace can be: begin$\langle \bot \rangle$ acquire$\langle \theta_1 \rangle$ acquire$\langle \theta_2 \rangle$ acquire$\langle \theta_1 \rangle$ release$\langle \theta_1 \rangle$ end$\langle \bot \rangle$ begin$\langle \bot \rangle$ acquire$\langle \theta_2 \rangle$ release$\langle \theta_2 \rangle$ end$\langle \bot \rangle$, where θ_1 maps r to r_1 and θ_2 maps r to r_2. We take the freedom to only list the parameter values when writing parameter instances, that is, $\langle r_1 \rangle$ instead of $\langle r \mapsto r_1 \rangle$, or $\tau\!\restriction_{r_2}$ instead of $\tau\!\restriction_{r \mapsto r_2}$, etc. With this notation, the above trace is: begin$\langle \rangle$ acquire$\langle r_1 \rangle$ acquire$\langle r_2 \rangle$ acquire$\langle r_1 \rangle$ release$\langle r_1 \rangle$ end$\langle \rangle$ begin$\langle \rangle$ acquire$\langle r_2 \rangle$ release$\langle r_2 \rangle$ end$\langle \rangle$. This trace involves two resources, r_1 and r_2, and really consists of *two trace slices*, one for each resource. The begin and end events belong to both trace slices. The slice corresponding to θ_1 is "begin acquire acquire release end begin end", while the one for θ_2 is "begin acquire end begin acquire release end". □

Definition 4. *Partial functions θ in $[X \rightharpoonup V]$ are called **parameter instances**. $\theta, \theta' \in [A \rightharpoonup B]$ are **compatible** if for any $x \in \mathsf{Dom}(\theta) \cap \mathsf{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible instances θ and θ', written $\theta \sqcup \theta'$, as follows:*

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ is defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\theta \sqcup \theta'$ *is also called the **least upper bound (lub)** of θ and θ'. θ' is **less informative** than θ, or θ is **more informative** than θ', written $\theta' \sqsubseteq \theta$, iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.*

For our example, $\langle \rangle$ is compatible with $\langle r_1 \rangle$ and with $\langle r_2 \rangle$, but $\langle r_1 \rangle$ and $\langle r_2 \rangle$ are not compatible; moreover, $\langle \rangle \sqsubseteq \langle r_1 \rangle$ and $\langle \rangle \sqsubseteq \langle r_2 \rangle$.

Definition 5. (Trace slicing) *Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ and θ in $[X \rightharpoonup V]$, let the θ-**trace slice** $\tau\!\restriction_\theta \in \mathcal{E}^*$ be the non-parametric trace defined as:*

- $\epsilon\!\restriction_\theta = \epsilon$, *where ϵ is the empty trace/word, and*
- $(\tau\, e\langle \theta' \rangle)\!\restriction_\theta = \begin{cases} (\tau\!\restriction_\theta)\, e & \text{when } \theta' \sqsubseteq \theta \\ \tau\!\restriction_\theta & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$

The trace slice $\tau\!\restriction_\theta$ first filters out all the parametric events that are not relevant for the instance θ, i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ, it forgets the parameters

so that the trace can be checked against base, non-parametric properties. It is crucial to discard parameter instances that are not relevant to θ during the slicing, including those more informative than θ, in order to achieve a "proper" slice for θ: in our running example, the trace slice for $\langle\rangle$ should contain only begin and end events and no acquire or release. Otherwise, the acquire and release of different resources will interfere with each other in the trace slice for $\langle\rangle$.

One should not confuse extracting/abstracting traces from executions with slicing traces. The former determines the events to include in the trace, as well as parameter instances carried by events, while the latter dispatches each event in the given trace to corresponding trace slices according to the event's parameter instance. Different abstractions may result in different parametric traces from the same execution and thus may lead to different trace slices for the same parameter instance θ. For the (map, collection, iterator) example in Section 1, $X = \{m, c, i\}$ and an execution may generate the following parametric trace: createColl$\langle m_1, c_1 \rangle$ createIter$\langle c_1, i_1 \rangle$ next$\langle i_1 \rangle$ updateMap$\langle m_1 \rangle$. The trace slice for $\langle m_1 \rangle$ is updateMap for this parametric trace. Now suppose that we are only interested in operations on maps. Then $X = \{m\}$ and the trace abstracted from the execution generating the above trace is createColl$\langle m_1 \rangle$ updateMap$\langle m_1 \rangle$, in which events and parameter bindings irrelevant to m are removed. Then the trace slice for $\langle m_1 \rangle$ is createColl updateMap. In this paper we focus on the trace slicing; more discussion about trace abstraction can be found in [10].

Definition 6. *Let X be a set of parameters together with their corresponding parameter values V, like in Definition 3, and let $P : \mathcal{E}^* \to \mathcal{C}$ be a non-parametric property like in Definition 2. Then we define the **parametric property** $\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X \rangle^*$ and categories $[[X \rightarrow V] \to \mathcal{C}]$)*

$$\Lambda X.P : \mathcal{E}\langle X \rangle^* \to [[X \rightarrow V] \to \mathcal{C}]$$

defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau\restriction_\theta)$ for any $\tau \in \mathcal{E}\langle X \rangle^*$ and any $\theta \in [X \rightarrow V]$. If $X = \{x_1, ..., x_n\}$ we may write $\Lambda x_1, ..., x_n.P$ instead of $(\Lambda\{x_1, ..., x_n\}.P$. Also, if P_φ is defined using a pattern or formula φ in some particular trace specification formalism, we take the liberty to write $\Lambda X.\varphi$ instead of $\Lambda X.P_\varphi$.

Parametric properties $\Lambda X.P$ over base properties $P : \mathcal{E}^* \to \mathcal{C}$ are therefore properties taking traces in $\mathcal{E}\langle X \rangle^*$ to categories $[[X \rightarrow V] \to \mathcal{C}]$, i.e., to function domains from parameter instances to base property categories. $\Lambda X.P$ is defined as if many instances of P are observed at the same time on the parametric trace, one property instance for each parameter instance, each property instance concerned with its events only, dropping the unrelated ones.

2.2 Algorithm for Online Parametric Trace Slicing

Definition 5 illustrates a way to slice a parametric trace for *given* parameter bindings. However, it is not suitable for online trace slicing, where the trace is observed incrementally and no future knowledge is available, because we cannot know all possible parameter instances θ apriori. We next define an algorithm

$\mathbb{A}\langle X \rangle$ in Fig. 1 that takes a parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ incrementally, and builds a partial function $\mathbb{T} \in [[X \rightarrow V] \rightarrow \mathcal{E}^*]$ of finite domain as a quick lookup table for all slices of τ.

Let us first introduce some operations on sets of partial functions used in $\mathbb{A}\langle X \rangle$ (only the informal intuition is given here; rigorous definitions can be found in [16]). Given sets of partial functions $\Theta, \Theta' \subseteq [X \rightarrow V]$, $\sqcup \Theta$ is the least informative partial function $\theta \in [X \rightarrow V]$ such that for any $\theta' \in \Theta$, $\theta' \sqsubseteq \theta$; $\max \Theta$ is the most informative $\theta \in \Theta$; $\Theta \sqcup \Theta' = \{\theta \sqcup \theta' \mid \theta \in \Theta, \theta' \in \Theta' \text{ s.t.}, \theta \sqcup \theta' \text{ exists}\}$; and $(\theta]_\Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta' \sqsubseteq$

```
Algorithm 𝔸⟨X⟩
Input: parametric trace τ ∈ 𝓔⟨X⟩*
Output: map 𝕋 ∈ [[X→V]→𝓔*] and
          set Θ ⊆ [X→V]
1  𝕋 ← ⊥; 𝕋(⊥) ← ε; Θ ← {⊥}
2  foreach e⟨θ⟩ in order in τ do
3  : foreach  θ′ ∈ {θ} ⊔ Θ  do
4  : : 𝕋(θ′) ← 𝕋(max (θ′]_Θ) e
5  : endfor
6  : Θ ← {⊥,θ} ⊔ Θ
7  endfor
```

Fig. 1. Parametric slicing algorithm $\mathbb{A}\langle X \rangle$

$\theta\}$. Note that $\sqcup \Theta$ and $\max \Theta$ may not exist. Then Theorem 1 shows that, for any $\theta \in [X \rightarrow V]$, the trace slice $\tau\!\!\restriction_\theta$ is $\mathbb{T}(\max (\theta]_\Theta)$ after $\mathbb{A}\langle X \rangle$ processes τ, where Θ is the domain of $\mathbb{T}$, calculated by $\mathbb{A}\langle X \rangle$ incrementally. Therefore, assuming that $\mathbb{A}\langle X \rangle$ is run on trace τ, all one has to do in order to calculate a slice $\tau\!\!\restriction_\theta$ for a given $\theta \in [X \rightarrow V]$ is to calculate $\max (\theta]_\Theta$ followed by a lookup into $\mathbb{T}$. This way the trace τ, which can be very long, is processed/traversed only once, as it is being generated, and appropriate data-structures are maintained by our algorithm that allow for retrieval of slices for any parameter instance θ, without traversing τ again.

Fig. 1 shows our trace slicing algorithm $\mathbb{A}\langle X \rangle$. In spite of $\mathbb{A}\langle X \rangle$'s small size, its proof of correctness is surprisingly intricate as shown in [16]. The algorithm $\mathbb{A}\langle X \rangle$ on input τ, written more succinctly $\mathbb{A}\langle X \rangle(\tau)$, traverses τ from its first event to its last event and, for each encountered event $e\langle \theta \rangle$, updates both its data-structures, $\mathbb{T}$ and Θ. After processing each event, the relationship between $\mathbb{T}$ and Θ is that the latter is the domain of the former. Line 1 initializes the data-structures: $\mathbb{T}$ is undefined everywhere (i.e., $\bot$) except for the undefined-everywhere function $\bot$, where $\mathbb{T}(\bot) = \epsilon$; as expected, Θ is then initialized to the set $\{\bot\}$. The code (lines 3 to 6) inside the outer loop (lines 2 to 7) can be triggered when a new event is received. When a new event is received, say $e\langle \theta \rangle$, $\mathbb{T}$ is updated as follows: for each $\theta' \in [X \rightarrow V]$ that can be obtained by combining θ with the compatible partial functions in the domain of the current $\mathbb{T}$, update $\mathbb{T}(\theta')$ by adding the non-parametric event e to the end of the slice corresponding to the largest (i.e., most "knowledgeable") entry in the current table $\mathbb{T}$ that is less informative or as informative as θ'; Θ is then extended in line 6.

Example. Consider the following sample parametric trace with events parametric in $\{a, b, c\}$: $\tau = e_1\langle a_1 \rangle\ e_2\langle a_2 \rangle\ e_3\langle b_1 \rangle\ e_4\langle a_2 b_1 \rangle\ e_5\langle a_1 \rangle\ e_6\langle \rangle\ e_7\langle b_1 \rangle$. Table 1 shows how $\mathbb{A}\langle X \rangle$ works on τ. An entry of the form "$\langle \theta \rangle : w$" in a table cell corresponding to a "current" parametric event $e\langle \theta \rangle$ means that $\mathbb{T}(\theta) = w$ after processing all the parametric events up to and including the current one; $\mathbb{T}$ is

Table 1. A run of the trace slicing algorithm $\mathbb{A}\langle X \rangle$

$e_1\langle a_1\rangle$	$e_2\langle a_2\rangle$	$e_3\langle b_1\rangle$	$e_4\langle a_2b_1\rangle$	$e_5\langle a_1\rangle$	$e_6\langle\rangle$	$e_7\langle b_1\rangle$
$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,\epsilon$	$\langle\rangle{:}\,e_6$	$\langle\rangle{:}\,e_6$
$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1$	$\langle a_1\rangle{:}\,e_1e_5$	$\langle a_1\rangle{:}\,e_1e_5e_6$	$\langle a_1\rangle{:}\,e_1e_5e_6$
	$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2$	$\langle a_2\rangle{:}\,e_2e_6$	$\langle a_2\rangle{:}\,e_2e_6$
		$\langle b_1\rangle{:}\,e_3$	$\langle b_1\rangle{:}\,e_3$	$\langle b_1\rangle{:}\,e_3$	$\langle b_1\rangle{:}\,e_3e_6$	$\langle b_1\rangle{:}\,e_3e_6e_7$
			$\langle a_1b_1\rangle{:}\,e_1e_3$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6$	$\langle a_1b_1\rangle{:}\,e_1e_3e_5e_6e_7$
			$\langle a_2b_1\rangle{:}\,e_2e_3$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6$	$\langle a_2b_1\rangle{:}\,e_2e_3e_4e_6e_7$

undefined on any other partial function. Obviously, the Θ corresponding to a cell is the union of all the θ's that appear in pairs "$\langle\theta\rangle : w$" in that cell. Trace slices for parameter instances, e.g., $\langle a_1b_1\rangle$ and $\langle a_2b_1\rangle$, which have *not* been seen in any observed event are also created. Note that, as each parametric event $e\langle\theta\rangle$ is processed, the non-parametric event e is added at most once to each slice. $\square$

$\mathbb{A}\langle X \rangle$ computes trace slices for all combinations of parameter instances observed in parametric trace events. Its complexity is therefore $O(n \times m)$ where n is the length of the trace and m is the number of all possible parameter combinations. However, $\mathbb{A}\langle X \rangle$ is not intended to be implemented directly; it is only used as a correctness backbone for other trace analysis algorithms, such as the monitoring algorithms discussed below. An alternative and apparently more efficient solution is to only record trace slices for parameter instances that actually appear in the trace (instead of for all combinations of them), and then construct the slice for a given parameter instance by combining such trace slices for compatible parameter instances. However, the complexity of constructing all possible trace slices at the end using such an algorithm is also $O(n \times m)$. In addition, $\mathbb{A}\langle X \rangle$ is more suitable for online monitoring: each event is sent to its slices (that are consumed by corresponding monitors) and never touched again.

$\mathbb{A}\langle X \rangle$ compactly and uniformly captures several special cases and subcases that are worth discussing. The discussion below can be formalized as an inductive (on the length of τ) proof of correctness for $\mathbb{A}\langle X \rangle$, but we prefer to keep this discussion informal here and use it as a means to better explain the algorithm $\mathbb{A}\langle X \rangle$, providing the reader with additional intuition for its difficulty and compactness. A rigorous proof can be found in [16].

Let us first note that a partial function added to Θ will never be removed from Θ; that's because $\Theta \subseteq \{\bot, \theta\} \sqcup \Theta$. The same holds true for the domain of $\mathbb{T}$, because line 4 can only add new elements to $\mathsf{Dom}(\mathbb{T})$; in fact, the domain of $\mathbb{T}$ is extended with precisely the set $\{\theta\} \sqcup \Theta$ after each event parametric in θ is processed by $\mathbb{A}\langle X \rangle$. Moreover, since $\mathsf{Dom}(\mathbb{T}) = \Theta = \Theta_\epsilon = \{\bot\}$ initially and since $\Theta \cup (\{\theta\} \sqcup \Theta) = \{\bot, \theta\} \sqcup \Theta$ while $\Theta_{\tau\, e\langle\theta\rangle} = \{\bot, \theta\} \sqcup \Theta_\tau$, where Θ_τ is Θ after $\mathbb{A}\langle X \rangle$ processes τ, we can inductively show that $\mathsf{Dom}(\mathbb{T}) = \Theta = \Theta_\tau$ each time after $\mathbb{A}\langle X \rangle$ is executed on a parametric trace τ. Each θ' considered by the loop at lines 3-5 has the property that $\theta \sqsubseteq \theta'$, and at (precisely) one iteration of the loop θ' is θ; indeed, $\theta \in \{\theta\} \sqcup \Theta$ because $\bot \in \Theta$. Essentially, the claimed Theorem 1 holds iff $\mathbb{T}(\theta') = \tau{\upharpoonright}_{\theta'}$ after $\mathbb{T}(\theta')$ is updated in line 4. A tricky observation which is crucial for this is that the updates of $\mathbb{T}(\theta')$ do not interfere with each other for different $\theta' \in \{\theta\} \sqcup \Theta$; otherwise the non-parametric event e may be added multiple times to some trace slices $\mathbb{T}(\theta')$.

Let us next informally argue, inductively, that it is indeed the case that $\mathbb{T}(\theta') = \tau\!\upharpoonright_{\theta'}$ after $\mathbb{T}(\theta')$ is updated in line 4 (it vacuously holds on the empty trace). Since $\max(\theta']_\Theta \in \Theta$, the inductive hypothesis tells us that $\mathbb{T}(\max(\theta']_\Theta)$ $= \tau\!\upharpoonright_{\max(\theta']_\Theta}$; these are further equal to $\tau\!\upharpoonright_{\theta'}$. Since $\theta \sqsubseteq \theta'$, the definition of trace slicing implies that $(\tau\, e\langle\theta\rangle)\!\upharpoonright_{\theta'} = \tau\!\upharpoonright_{\theta'}\, e$. Therefore, $\mathbb{T}(\theta')$ is indeed $(\tau\, e\langle\theta\rangle)\!\upharpoonright_{\theta'}$ after line 4 of $\mathbb{A}\langle X\rangle$ is executed while processing the event $e\langle\theta\rangle$ that follows trace τ. This concludes our informal proof sketch.

Let $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}$ and $\mathbb{A}\langle X\rangle(\tau).\Theta$ be $\mathbb{T}$ and Θ of $\mathbb{A}\langle X\rangle$ after it processes τ.

Theorem 1. *The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:*

1. *$Dom(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X\rangle(\tau).\Theta = \Theta_\tau$;*
2. *$\mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta) = \tau\!\upharpoonright_\theta$ for any $\theta \in \mathbb{A}\langle X\rangle(\tau).\Theta$;*
3. *$\tau\!\upharpoonright_\theta = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta})$ for any $\theta \in [X\!\rightarrow\!V]$.*

3   Online Parametric Trace Monitoring

Here we first define monitors M and parametric monitors $\Lambda X.M$. Like for parametric properties, which are just properties over parametric traces, we show that parametric monitors are also just monitors, but for parametric events and with instance-indexed states and output categories. We show that a parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with P the property monitored by M. Finally, we present an online monitoring algorithm based on algorithm $\mathbb{A}\langle X\rangle$ and then refine it to an efficient monitoring algorithm.

3.1   Monitors and Parametric Monitors

Non-parametric monitors are defined as a variant of Moore machines:

Definition 7. *A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, \imath, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow \mathcal{C})$, where S is a set of states, $\mathcal{E}$ is a set of input events, $\mathcal{C}$ is a set of output categories, $\imath \in S$ is the initial state, σ is the transition function, and γ is the output function. The transition function is extended to $\sigma : S \times \mathcal{E}^* \rightarrow S$ the standard way.*

The notion of a monitor above is rather conceptual. Actual implementations of monitors need not generate all the state space apriori, but on a "by need" basis. Allowing monitors with infinitely many states is a necessity in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many. For example, monitors for context-free grammars like the ones in [15] have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well.

Definition 8. *$M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \rightarrow \mathcal{C}$ iff $\gamma(\sigma(\imath, w)) = P(w)$ for each $w \in \mathcal{E}^*$. Every monitor M defines the property $\mathcal{P}_M$: $\mathcal{E}^* \rightarrow \mathcal{C}$ with $\mathcal{P}_M(w) = \gamma(\sigma(\imath, w))$; note that M is a monitor for $\mathcal{P}_M$. Monitors M and M' are **equivalent**, written $M \equiv M'$ iff $\mathcal{P}_M = \mathcal{P}_{M'}$.*

Proposition 1. *Every property P defines a monitor $\mathcal{M}_P$ with $\mathcal{M}_P$ a monitor for P. For any property P, $\mathcal{P}_{\mathcal{M}_P} = P$. For any monitor M, if $M = \mathcal{M}_P$ for some property P then $\mathcal{M}_{\mathcal{P}_M} \equiv M$.*

We next define parametric monitors in the same style as the other parametric entities defined in this paper: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 9. *Given parameters X with corresponding values V and monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, the **parametric monitor** $\Lambda X.M$ is the monitor $([[X \to V] \to S], \mathcal{E}\langle X \rangle, [[X \to V] \to \mathcal{C}], \lambda\theta.\iota, \Lambda X.\sigma, \Lambda X.\gamma)$, with $\Lambda X.\sigma : [[X \to V] \to S] \times \mathcal{E}\langle X \rangle \to [[X \to V] \to S]$ and $\Lambda X.\gamma : [[X \to V] \to S] \to [[X \to V] \to \mathcal{C}]$ defined as*

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases}$$
$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

for any $\delta \in [[X \to V] \to S]$ and any $\theta, \theta' \in [X \to V]$.

Therefore, a state δ of parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ, takes parametric events as input, and outputs categories indexed by parameter instances (one category of M per instance).

Proposition 2. *If M is a monitor for P, then $\Lambda X.M$ is a monitor for parametric property $\Lambda X.P$, or, $\mathcal{P}_{\Lambda X.M} = \Lambda X.\mathcal{P}_M$.*

3.2   Algorithm for Online Parametric Trace Monitoring

We next propose a monitoring algorithm for parametric properties. A first challenge here is how to represent the states of the parametric monitor. Inspired by algorithm $\mathbb{A}\langle X \rangle$, we encode functions $[[X \to V] \to S]$ as tables with entries indexed by parameter instances in $[X \to V]$ and with contents states in S. Such tables will have finite entries since each event binds only a finite number of parameters. Fig. 2 shows our monitoring algorithm for parametric properties. Given parametric property $\Lambda X.P$ and M a monitor for P, $\mathbb{B}\langle X \rangle(M)$ yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$. Section 4 shows one way to use this algorithm: a monitor M is first synthesized from the base property P, then that monitor M is used to synthesize the monitor $\mathbb{B}\langle X \rangle(M)$ for the parametric property, $\Lambda X.P$.

$\mathbb{B}\langle X \rangle(M)$ follows very closely the algorithm for trace slicing in Fig. 1, the main difference being that trace slices are processed, as generated, by M: instead of calculating the trace slice of θ' by appending base event e to the corresponding existing trace slice in line 4 of $\mathbb{A}\langle X \rangle$, we now calculate and store in table Δ the state of the "monitor instance" corresponding to θ' by sending e to the corresponding existing monitor instance (line 4 in $\mathbb{B}\langle X \rangle(M)$); at the same time we also calculate the output corresponding to that monitor instance and store it in table Γ. In other words, we replace trace slices in $\mathbb{A}\langle X \rangle$ by local

monitors processing those slices. We also check whether $\Gamma(\theta')$ at line 5 violates or validates the property and, if so, a message including θ' is output. Given a monitor M, let $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ be the monitor defined by $\mathbb{B}\langle X\rangle(M)$. Theorem 2 then proves the correctness of $\mathbb{B}\langle X\rangle$.

Theorem 2. $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} \equiv \Lambda X.M$ *for any monitor M. If M is a monitor for P, then $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ is a monitor for parametric property $\Lambda X.P$.*

Algorithm $\mathbb{B}\langle X\rangle(M=(S,\mathcal{E},\mathcal{C},\imath,\sigma,\gamma))$
Input: parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$
Output:$\Gamma : [[X{\rightarrow}V]{\rightarrow}\mathcal{C}]$ and $\Theta \subseteq [X{\rightarrow}V]$
1 $\Delta \leftarrow \bot;\ \Delta(\bot) \leftarrow \imath;\ \Theta \leftarrow \{\bot\}$
2 foreach $e\langle\theta\rangle$ *in order in* τ do
3 : foreach $\theta' \in \{\theta\} \sqcup \Theta$ do
4 : : $\Delta(\theta') \leftarrow \sigma(\Delta(\max(\theta']_\Theta),e)$
5 : : $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$
6 : endfor
7 : $\Theta \leftarrow \{\bot,\theta\} \sqcup \Theta$
8 endfor

Fig. 2. Monitoring algorithm $\mathbb{B}\langle X\rangle$

3.3   Optimized Online Monitoring Algorithm

Algorithm $\mathbb{C}\langle X\rangle$ in Fig. 3 refines Algorithm $\mathbb{B}\langle X\rangle$ in Fig. 2 for efficient online monitoring. $\mathbb{C}\langle X\rangle$ essentially expands the body of the outer loop in $\mathbb{B}\langle X\rangle$ (lines 3 to 7 in Fig. 2). The direct use of $\mathbb{B}\langle X\rangle$ would yield prohibitive runtime overhead when monitoring large traces, because its inner loop requires search for all parameter instances in Θ that are compatible with θ; this search can be very expensive. $\mathbb{C}\langle X\rangle$ introduces an auxiliary data structure and illustrates a mechanical way to accomplish the search, which also facilitates further optimizations.

Algorithm $\mathbb{C}\langle X\rangle(M = (S,\mathcal{E},\mathcal{C},\imath,\sigma,\gamma))$
Globals: mappings $\Delta : [[X{\rightarrow}V]{\rightarrow}S]$,
       $\Gamma : [[X{\rightarrow}V]{\rightarrow}\mathcal{C}]$,
       $\mathcal{U} : [X{\rightarrow}V] \rightarrow \mathcal{P}_f([X{\rightarrow}V])$
Initialization:
  $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X{\rightarrow}V]$,
  $\Delta(\bot) \leftarrow \imath$
function main($e\langle\theta\rangle$)
 1 if $\Delta(\theta)$undefined then
 2 : foreach $\theta_{max} \sqsubset \theta$ (in reversed
   : topological order) do
 3 : : if $\Delta(\theta_{max})$ defined then
 4 : : : goto 7
 5 : : endif
 6 : endfor
 7 : defineTo(θ,θ_{max})
 8 : foreach $\theta_{max} \sqsubset \theta$ (in reversed
   : topological order) do

 9 : : foreach $\theta_{comp} \in \mathcal{U}(\theta_{max})$
   : : compatible with θ do
10 : : : if $\Delta(\theta_{comp} \sqcup \theta)$ undef. then
11 : : : : defineTo($\theta_{comp} \sqcup \theta,\theta_{comp}$)
12 : : : endif
13 : : endfor
14 : endfor
15 endif
16 foreach $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ do
17 : $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'),e)$
18 : $\Gamma(\theta') \leftarrow \sigma(\Delta(\theta'))$
19 endfor
function defineTo(θ,θ')
 1 $\Delta(\theta) \leftarrow \Delta(\theta')$
 2 foreach $\theta'' \sqsubset \theta$ do
 3 : $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
 4 endfor

Fig. 3. Monitoring algorithm $\mathbb{C}\langle X\rangle$

$\mathbb{C}\langle X \rangle$ uses three tables: Δ, $\mathcal{U}$ and Γ. Δ and Γ are the same as Δ and Γ in $\mathbb{B}\langle X \rangle$, respectively. $\mathcal{U}$ is an auxiliary data structure used to optimize the search "for all $\theta' \in \{\theta\} \sqcup \Theta$" in $\mathbb{B}\langle X \rangle$ (line 3 in Fig. 2). It maps each parameter instance θ into the finite set of parameter instances encountered in Δ so far that are strictly more informative than θ, i.e., $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \mathsf{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$. Another major difference between $\mathbb{B}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ is that $\mathbb{C}\langle X \rangle$ does *not* maintain Θ explicitly: Θ at the beginning/end of the body of the outer loop in $\mathbb{B}\langle X \rangle$ is $\mathsf{Dom}(\Delta)$ at the beginning/end of $\mathbb{C}\langle X \rangle$, respectively. However, Θ is fixed during the loop at lines 3 to 6 in $\mathbb{B}\langle X \rangle$ and updated atomically in line 7, while $\mathsf{Dom}(\Delta)$ can be changed at any time during the execution of $\mathbb{C}\langle X \rangle$.

$\mathbb{C}\langle X \rangle$ is composed of two functions, main and defineTo. The defineTo function takes two parameter instances, θ and θ', and adds a new entry corresponding to θ into Δ and $\mathcal{U}$. Specifically, it sets $\Delta(\theta)$ to $\Delta(\theta')$ and adds θ into the set $\mathcal{U}(\theta'')$ for each $\theta'' \sqsubset \theta$. The main function differentiates two cases when a new event $e\langle\theta\rangle$ is received and processed. The simpler case is that Δ is already defined on θ, i.e., $\theta \in \Theta$ at the beginning of the iteration of the outer loop in $\mathbb{B}\langle X \rangle$. In this case, $\{\theta\} \sqcup \Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta \sqsubseteq \theta'\} \subseteq \Theta$, so the lines 3 to 6 in $\mathbb{B}\langle X \rangle$ become precisely the lines 16 to 19 in $\mathbb{C}\langle X \rangle$. In the other case, when Δ is not already defined on θ, main takes two steps to handle e. The first step searches for new parameter instances introduced by $\{\theta\} \sqcup \Theta$ and adds entries for them into Δ (lines 2 to 15). We first add an entry to Δ for θ at lines 2 to 7. Then we search for all parameter instances θ_{comp} that are compatible with θ, making use of $\mathcal{U}$ (line 8 and 9); for each such θ_{comp}, an appropriate entry is added to Δ for its lub with θ, and $\mathcal{U}$ updated accordingly (lines 10 to 12). This way, Δ will be defined on all the new parameter instances introduced by $\{\theta\} \sqcup \Theta$ after the first step. In the second step, the related monitor states and outputs are updated in a similar way as in the first case (lines 16 to 19). It is interesting to note how $\mathbb{C}\langle X \rangle$ searches at lines 2 and 8 for $\max(\theta]_\Theta$ that $\mathbb{B}\langle X \rangle$ refers to at line 4 in Fig. 2: it enumerates all the $\theta_{max} \sqsubset \theta$ in reversed topological order (larger to smaller); we proved that $\max(\theta]_\Theta$ exists and this search will find it ([16]).

Correctness of $\mathbb{C}\langle X \rangle$. We next argue informally the correctness of $\mathbb{C}\langle X \rangle$ (formal proofs can be found in [16]) by showing that it is equivalent to the body of the outer loop in $\mathbb{B}\langle X \rangle$. Suppose that parametric trace τ has already been processed by both $\mathbb{C}\langle X \rangle$ and $\mathbb{B}\langle X \rangle$, and a new event $e\langle\theta\rangle$ is to be processed next. First, note that $\mathbb{C}\langle X \rangle$ terminates: there is only a finite number of partial maps less informative than θ, that is, only a finite number of iterations for the loops at lines 2 and 8 in main; since $\mathcal{U}$ is only updated at line 3 in defineTo, $\mathcal{U}(\theta)$ is finite for any $\theta \in [X \rightarrow V]$ and thus the loop at line 9 in main also terminates. Assuming that running the base monitor M takes constant time, the worse case complexity of $\mathbb{C}\langle X \rangle(M)$ is $O(k \times l)$ to process $e\langle\theta\rangle$, where k is $2^{|\mathsf{Dom}(\theta)|}$ and l is the number of incompatible parameter instances in τ. Parametric properties often have a fixed and small number of parameters, in which case k is not significant. Depending on the trace, l can grow arbitrarily large; in the worst case, each event may carry an instance incompatible with the previous ones.

Next result establishes the correctness of $\mathbb{C}\langle X \rangle$. Fix a monitor M. Let $\Delta_{\mathbb{C}}^b$ and $\Gamma_{\mathbb{C}}^b$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ in $\mathbb{C}\langle X \rangle(M)$ begins ("b"="begin"); let $\Delta_{\mathbb{C}}^e$ and $\Gamma_{\mathbb{C}}^e$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ ends ("e"="end"); let $\mathbb{B}\langle X \rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X \rangle(M)(\tau).\Gamma$ be the Δ and Γ after $\mathbb{B}\langle X \rangle(M)$ processes trace τ.

Theorem 3. *If* $\Delta_{\mathbb{C}}^b = \mathbb{B}\langle X \rangle(M)(\tau).\Delta$ *and* $\Gamma_{\mathbb{C}}^b = \mathbb{B}\langle X \rangle(M)(\tau).\Gamma$, *then* $\Delta_{\mathbb{C}}^e = \mathbb{B}\langle X \rangle(M)(\tau\,e\langle\theta\rangle).\Delta$ *and* $\Gamma_{\mathbb{C}}^b = \mathbb{B}\langle X \rangle(M)(\tau\,e\langle\theta\rangle).\Gamma$.

4   Implementation and Evaluation

We have implemented our online monitoring algorithm in a prototype, here called PMon (from "Parametric Monitoring"), and evaluated it on the DaCapo benchmark [6]. Some optimizations have also been implemented. Note that $\mathbb{C}\langle X \rangle$ iterates through all the possible parameter instances that are less informative than θ in three different loops: at lines 2 and 8 in main, and at line 2 in $\mathsf{defineTo}$. Hence, it is important to reduce the number of such instances in each loop. A static analysis of the specification, discussed in [8], exhaustively explores all possible event combinations that can lead to violations of the property, and then the number of loop iterations is reduced by skipping parameter instances that cannot affect the result of monitoring. The static analysis is used at compile time to unroll the loops in $\mathbb{C}\langle X \rangle$ and reduce the size of Δ and $\mathcal{U}$.

Another optimization is based on the observation that the monitoring process needs to start only when certain events are received. Such events are called monitor creation events in [9]. The parameter instances carried by such creation events may also be used to reduce the number of parameter instances that need to be considered. An extreme, yet surprisingly common case is when creation events instantiate all the property parameters. In this case, the monitoring process does not need to search for compatible parameter instances even when an event with an incomplete parameter instance is observed. The current JavaMOP [9] supports only traces whose monitoring starts with a fully instantiated creation event; this was perceived as an inherent limitation of JavaMOP, a consequence of its generality [3]. Interestingly, it now becomes just a common-case optimization of our novel, general and unrestricted technique presented here.

Experiments and Evaluation. The following properties from [8] were checked in our experiments. The latter two cannot be handled by JavaMOP.

- LeakingSync. Only access a synchronized collection using its synchronized wrapper. One violation pattern monitored: $\Lambda c.\ \mathsf{sync}(c)\ \mathsf{asyncAccess}(c)$.
- FailSafeEnum. Do not update a vector while enumerating over it. The following violation pattern monitored: $\Lambda v, e.\ \mathsf{createEnum}(v, e)\ \mathsf{modify}(v)\ \mathsf{access}(e)$.
- ASyncIterCol. Only iterate a synchronized collection c when holding a lock on c. Two violation patterns monitored: $\Lambda c, i.\ \mathsf{sync}(c)\ \mathsf{ayncCreateIter}(c, i)$ and $\Lambda c, i.\ \mathsf{sync}(c)\ \mathsf{syncCreateIter}(c, i)\ \mathsf{asyncAccess}(i)$.
- ASyncIterMap. Only iterate a synchronized map m when holding a lock on m. Two violation patterns monitored: $\Lambda m, s, i.\ \mathsf{sync}(m)\ \mathsf{getSet}(m, s)\mathsf{async}$-$\mathsf{CreateIter}(s, i)$, $\Lambda m, s, i.\ \mathsf{sync}(m)\ \mathsf{getSet}(m, s)\ \mathsf{syncCreateIter}(s, i)\ \mathsf{asyncAccess}(i)$.

Table 2. Average percent runtime overhead for PMon, manually coded monitors(Man), Tracematches(TM) and JavaMOP(MOP), with convergence within 3%. *: Cannot be handled by JavaMOP

	LeakingSync				FailSafeEnum				ASyncIterCol*			ASyncIterMap*		
	PMon	Man	TM	MOP	PMon	Man	TM	MOP	PMon	Man	TM	PMon	Man	TM
antlr	2	1	5	2	0	0	1	0	1	0	0	2	1	2
bloat	140	145	785	141	0	2	0	2	721	150	1459	660	164	2300
chart	25	21	70	24	1	0	0	0	2	0	0	0	0	0
eclipse	0	0	0	0	0	0	0	2	1	2	0	1	0	0
fop	53	47	146	50	1	0	0	0	2	1	1	2	1	0
hsqldb	2	5	24	4	0	0	25	0	1	0	25	1	0	0
jython	62	52	55	59	0	0	8	0	0	0	9	0	0	9
luindex	8	7	20	7	7	4	16	3	3	0	2	1	0	4
lusearch	12	10	52	12	5	2	28	7	4	1	9	0	0	8
pmd	55	47	53	52	0	0	0	0	37	30	36	50	49	53
xalan	39	29	117	40	5	6	33	4	1	1	6	1	1	7

These properties were chosen since they involve some of the most used data structures in Java and generate intensive monitoring overhead; also, their overhead is a consequence of the huge number of parameter instances to handle and *not* because of the complexity of the base, non-parametric properties.

Using the above properties, we compared our implementation with three other monitoring approaches, namely: optimal manually implemented monitors[1], Tracematches and JavaMOP. The latter two are chosen for comparison because they are very efficient monitoring systems [3,9]. The evaluation carried out on a 1.5GB, Pentium 4 2.66GHz machine running Ubuntu 7.10. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs, as shown in Table 2. The provided default input was used with the -converge option to execute the benchmark multiple times until the execution time falls within 3% variation. The average execution time of six iterations after convergence is used.

The results are shown in Table 2. Among all 44 experiments, PMon generates 14% runtime overhead on average with more than 15% in only 10 experiments, showing that our algorithm is efficient[2]. Comparing with other approaches, we have the following observations: 1) PMon performed as well as or better than Tracematches in all cases, although the latter has domain specific optimizations for its hard-wired parametric regular patterns; 2) PMon generates similar runtime overhead as JavaMOP in the cases that can be handled by JavaMOP, showing that PMon conservatively extends the limited algorithm implemented in JavaMOP; 3) the monitoring code generated by PMon performs as well as the manually implemented monitors in most cases in the evaluated properties.

5   Related Work

Several approaches have been proposed to specify and monitor parametric properties. Tracematches [1,3] is an extension of AspectJ [2] supporting specifications

---

[1] Borrowed from [8], supposedly the best monitoring code for the given properties.

[2] These properties generated a tremendous number of events and parameter instances, e.g., millions of events and instances seen for LeakingSync and FailSafeEnum [9].

of parametric regular patterns; when patterns are matched during the execution, user-defined advice can be triggered. J-LO [7] is a variation of Tracematches that supports linear temporal logic properties. Also based on AspectJ, [13] proposes Live Sequence Charts (LSC) [11] as an inter-object scenario-based specification formalism; LSC is implicitly parametric, requiring dynamic parameter binding at runtime. Tracematches, J-LO and LSC [13] support a limited number of parameters, and each handles parameterization in a way that is specific to its particular specification formalism. Our proposed technique is generic in the specification formalism, and admits a potentially unlimited number of parameters.

Program Query Language (PQL) [14] allows the specification and monitoring of parametric context-free grammar (CFG) patterns. Unlike previous approaches, PQL can associate parameters with sub-patterns that can be recursively matched at runtime, yielding a potentially unbounded number of parameters. PQL's approach to parametric monitoring is specific to its particular CFG-based specification formalism. Also, PQL's design does not support arbitrary execution traces. For example, field updates and method begins are not observable. Like PQL, our technique also allows an unlimited number of parameters. Unlike PQL, our technique is not limited to particular events and is generic in the property specification formalism; CFGs are just one such possible formalism.

Eagle [4], RuleR [5], and Program Trace Query Language (PTQL) [12] are very general trace specification and monitoring systems, whose specification formalisms allow complex properties with parameter bindings anywhere in the specification. Eagle and RuleR are based on fixed-point logics and rewrite rules, while PTQL is based on SQL relational queries. These systems attempt to define general specification formalisms supporting data binding among many other features, while we attempt to define a general parameterization approach that is logic-independent. The very general specification formalisms tend to be slower [3,15,9]. We believe that our techniques can be used as an optimization for certain common types of properties expressible in these systems: use any of these to specify the base property P, then use our generic techniques to analyze $\Lambda X.P$.

6   Concluding Remarks and Future Work

A semantic foundation for parametric traces, properties and monitoring was proposed. A parametric trace slicing technique, which was discussed and proved correct, allows the extraction of all the non-parametric trace slices from a parametric slice by traversing the original trace only once and dispatching each parametric event to its corresponding slices. It thus enables the leveraging of any non-parametric, i.e., conventional, trace analysis techniques to the parametric case. A parametric monitoring technique then makes use of it to monitor arbitrary parametric properties against parametric execution traces using and indexing ordinary monitors for the base, non-parametric property. An implementation of the discussed techniques reveals that their generality, compared to the existing similar, but limited, techniques, does not come at a performance expense.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005. ACM, New York (2005)
2. AspectJ, `http://eclipse.org/aspectj/`
3. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitoring feasible. In: OOPSLA 2007. ACM, New York (2007)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
5. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: From Eagle to RuleR. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007)
6. 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., Van Drunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006. ACM Press, New York (2006)
7. Bodden, E.: J-lo, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (2005)
8. Bodden, E., Chen, F., Roşu, G.: Dependent advice: A general approach to optimizing history-based aspects. In: AOSD 2009. ACM, New York (2009)
9. Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: OOPSLA 2007. ACM, New York (2007)
10. Chen, F., Roşu, G.: Mining Parametric State-Based Specifications from Executions. Technical Report UIUCDCS-R-2008-3000, Dept. of Computer Science at UIUC (2008)
11. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)
12. Goldsmith, S., O'Callahan, R., Aiken, A.: Relational queries over program traces. In: OOPSLA 2005. ACM Press, New York (2005)
13. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling lscs into aspectj. In: FSE 2006, pp. 219–230. ACM, New York (2006)
14. Martin, M., Livshits, V.B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: OOPSLA 2005. ACM, New York (2005)
15. Meredith, P., Jin, D., Chen, F., Roşu, G.: Efficient monitoring of parametric context-free patterns. In: ASE 2008. IEEE/ACM (2008)
16. Roşu, G., Chen, F.: Parametric Trace Slicing and Monitoring. Technical Report UIUCDCS-R-2008-2977, Dept. of Computer Science at UIUC (2008)

From Tests to Proofs

Ashutosh Gupta[1], Rupak Majumdar[2], and Andrey Rybalchenko[1]

[1] Max Planck Institute for Software Systems
[2] University of California, Los Angeles

Abstract. We describe the design and implementation of an automatic invariant generator for imperative programs. While automatic invariant generation through constraint solving has been extensively studied from a theoretical viewpoint as a classical means of program verification, in practice existing tools do not scale even to moderately sized programs. This is because the constraints that need to be solved even for small programs are already too difficult for the underlying (non-linear) constraint solving engines. To overcome this obstacle, we propose to strengthen static constraint generation with information obtained from static abstract interpretation and dynamic execution of the program. The strengthening comes in the form of additional linear constraints that trigger a series of simplifications in the solver, and make solving more scalable. We demonstrate the practical applicability of the approach by an experimental evaluation on a collection of challenging benchmark programs and comparisons with related tools based on abstract interpretation and software model checking.

1 Introduction

Programmers make mistakes, and much time and effort is spent on finding and fixing these mistakes. While it has long been known that *program invariants* are the key to proving a program correct with respect to a safety property [10, 17], their applicability has been limited in practice since they often require explicit and expensive programmer annotations. To circumvent this problem, there has been considerable research effort in program analysis for *automatic* inference of program invariants [1, 2, 4, 16, 27]. In these algorithms, a set of constraints is generated from the program text whose solution provides an inductive invariant proof of program correctness.

In the *abstract interpretation* based approach [4, 7, 24] to inductive invariant inference, one computes the fixpoint of the program semantics relative to an abstract domain. In case the abstract domain has infinite height (for example, the domain of polyhedra), termination of the fixpoint computation is enforced by a widening operator. In the *counterexample-guided abstraction refinement (CEGAR)* approach [1, 16], one starts with a set of predicates, and uses spurious counterexamples produced by model checking to dynamically discover new predicates that serve as building blocks for the proof of program correctness. Finally, in the *constraint-based approach* [5, 14, 27], a parametric representation of an invariant map serves a starting point. Then, inductiveness and safety conditions are encoded as constraints on the parameters. Once these constraints have been determined, any satisfying assignment is guaranteed to yield an inductive invariant of the program. For example, an invariant template in linear arithmetic will

Table 1. Comparison of invariant-based verification tools on benchmark problems

File	State-of-the-art techniques				This paper
	INTERPROC	BLAST	INVGEN	INVGEN+Z3	
Seq	×	diverge	23s	1s	**0.5s**
Seq-z3	×	diverge	23s	9s	**0.5s**
Seq-len	×	diverge	T/O	T/O	**2.8s**
nested	×	**1.2s**	T/O	T/O	2.3s
svd(light)	×	50s	T/O	T/O	**14.2s**
heapsort	×	**3.4s**	T/O	T/O	13.3s
mergesort	×	**18s**	T/O	52s	170s
SpamAssassin-loop*	✓	22s	T/O	5s	**0.4s**
apache-get-tag*	×	5s	**0.4s**	10s	0.7s
sendmail-fromqp*	×	diverge	0.3s	5s	**0.3s**

specify for each program point an expression of the form $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n \leq 0$, where $x_1, \ldots, x_n$ are program variables, and $\alpha_0, \ldots, \alpha_n$ are unknown parameters. The control flow graph of the program will specify constraints on the parameters at each program point, such that a global solution for all the α's produces an invariant.

While these techniques hold the potential for extremely sophisticated reasoning about programs, each technique by itself often fails to verify programs, since in practice reasoning about correctness often requires combining the strength of each individual approach. In this paper, we demonstrate the potential of such a combination. We describe the design and implementation of a constraint-based invariant generator for linear arithmetic invariants. In our implementation, we use information from static abstract interpretation-based techniques as well as from dynamic testing to aggressively simplify constraints. Our experimental results demonstrate that using these optimizations our invariant generator can automatically verify many problems for which all the existing approaches we tried are unsuccessful.

It is important to mention that for each of our examples there is (in theory) a polyhedral abstract domain equipped with a suitable widening operator that can successfully prove the desired assertion. Our approach targets the cases for which the *existing* abstract interpreters fail due to heuristic choices made in the implementation that trade off precision for speed. For example, Figure 1(a) shows a program from [13] for which an abstract interpreter implementing the standard convex hull-based widening cannot prove the assertion. In our experiments, the abstract interpretation tool INTERPROC finds the invariants $z = 10w$ and $y \leq 100x$ at line 2 but not the crucial $y \geq x$. We observed that our approach finds the missing fact $y \geq x$ which together with the invariants found by INTERPROC, is sufficient to prove the assertion.

Table 1 shows the results of running a collection of state-of-the-art program verification tools on a set of common benchmark programs for software verification, including some challenge programs from [21], which are marked with the star symbol "*". INTERPROC [22] is a tool based on abstract interpretation (we used the PPL library together with the octagon domain when applying INTERPROC). BLAST [16] is a software model checker based on counterexample refinement. INVGEN is our previous implementation of constraint-based invariant generation using constraint logic programming

(CLP) as a constraint solver [2]. INVGEN+Z3 is the same constraint-based invariant generator but using the Z3 decision procedure [8] as the constraint solver, which applies the Boolean satisfiability-based encoding proposed in [14]. As is evident from Table 1, the results we obtained for the existing tools on the benchmark examples are disappointing. In Column 2, there is a "×" mark for each program for which INTERPROC was too imprecise to verify the assertion. In Column 3, the counterexample refinement procedure of Blast diverges on several examples. In Columns 4 and 5, the invariant generation procedures time out, denoted by "T/O", on most examples as the constraints become too hard to solve (both for CLP and for SAT). In contrast, our technique is able to efficiently solve all the examples, as shown in the last column.

While our invariant generator can be used in isolation, we have also integrated it with the Blast software model checker and have used it as the counterexample refinement engine using path programs [3]. Invariants for path programs provide additional predicates that refine the abstraction for the software model checker, and can produce better refinement predicates than usually available with current techniques, e.g. [15]. Software model checkers with path program-based counterexample analysis are well-suited for our techniques because they (automatically) generate small program units to either test for bugs or provide invariants. Using this integration, we have applied our implementation to verify a set of software verification benchmark programs [21] recently introduced as a challenge to the community. The examples in the benchmark set are extracted from common security-critical code, and contain assertions related to buffer bounds checking. Our implementation was able to verify all the (correct) programs in the benchmark in about 10s of total time.

Related Work. Our work is influenced by recent advances in automatic static inference of inductive invariants using constraint solving [6, 14, 26] as well as by the use of dynamic analysis to estimate and infer likely system properties [9].

Constraint-based invariant synthesis techniques using templates in linear [2, 5, 14] and polynomial [20, 26] arithmetic have been extensively studied, but their application has been limited by the cost of the constraint solving process. As we demonstrate in our experiments, even on quite small examples the constraint solver is likely to time-out. Our static and dynamic constraint simplification techniques limit the search space for the constraint solvers. Our experiments demonstrate orders of magnitude improvements over existing making it feasible to apply these techniques to larger programs.

Software model checking tools, e.g. [1, 16, 19], have previously used invariants from abstract interpretation—most notably alias analysis, but also octagonal constraints [19]—to strengthen the transition relation of the program. The contribution of this work to the research on software model checking is a powerful predicate inference engine using invariant generation. We also perform detailed comparisons of the benefits of combining invariant generation with abstract interpretation, as well as combining invariant generation with CEGAR-based software verification.

Pure dynamic analysis has been used to identify likely, but not necessarily correct, program invariants [9]. The technique uses program tests to evaluate candidate predicates from some a priori fixed database. The predicates that evaluate to true on all test runs are returned as likely invariants. The basic technique is not sound, as the test suite could be inadequate. Hence in a second step, the inferred invariants are provided to a

```
1   int x=0; y=0; z=0 w=0;
2   while(*){
3    if(*){
4      x++; y+=100;                    1   int i,j,k,n,m;
5    }else if(*){                      2
6      if (x>=4){ x++; y++; }          3   assume(n<=m);
7    }else if(y>10*w && z>=100*x){     4   for (i=0;i<n;i++)
8      y=-y;                           5    for (j=0;j<n;j++)
9    }                                 6     for (k=j; k<n+m;k++)
10   w++; z+=10;                       7       assert(i+j<=n+k+m);
11  }
12  if( x>=4 && y <=2) error();
```

| (a) | (b) |

Fig. 1. (a) Example from [13]. (b) Example `nested.c`.

verification-condition based program verifier. If the verifier succeeds, the combination of the dynamic step and the verification ensures program safety, while removing the need for providing manual invariants. However, there are some shortcomings of this technique. First, since the predicates are chosen from some fixed set (usually for efficiency in evaluation), the required program invariants may not fall into this fixed class. Second, the generated invariants are not in general inductive, therefore if the verifier fails, it is not evident if either a guessed invariant is wrong (that is, more tests should be generated to remove it from the discovered set), or if the guessed invariant does represent all reachable states, but is too weak to allow the verifier to complete the proof.

2   Example

We illustrate our idea using the example program `nested.c` shown in Figure 1(b). We want to construct an invariant that proves the assertion in line 7.

The core idea of our tool is to perform constraint-based invariant synthesis. Our algorithm automatically discovers, through an iterative process, that we need an invariant templates to be a conjunction of four inequalities for each loop head. The invariants for intermediate locations (between loop heads) can be computed from assertions for these locations by propagating strongest postconditions (or weakest preconditions). For clarity of presentation, we shall only show details relevant to the first conjunct in each template. We use the template map η such that

$$\eta.4 \;=\; \alpha + \alpha_{\mathrm{i}}\mathrm{i} + \alpha_{\mathrm{j}}\mathrm{j} + \alpha_{\mathrm{k}}\mathrm{k} + \alpha_{\mathrm{m}}\mathrm{m} + \alpha_{\mathrm{n}}\mathrm{n} \leq 0 \wedge \ldots \wedge \ldots \wedge \ldots,$$
$$\eta.5 \;=\; \beta + \beta_{\mathrm{i}}\mathrm{i} + \beta_{\mathrm{j}}\mathrm{j} + \beta_{\mathrm{k}}\mathrm{k} + \beta_{\mathrm{m}}\mathrm{m} + \beta_{\mathrm{n}}\mathrm{n} \leq 0 \wedge \ldots \wedge \ldots \wedge \ldots,$$
$$\eta.6 \;=\; \gamma + \gamma_{\mathrm{i}}\mathrm{i} + \gamma_{\mathrm{j}}\mathrm{j} + \gamma_{\mathrm{k}}\mathrm{k} + \gamma_{\mathrm{m}}\mathrm{m} + \gamma_{\mathrm{n}}\mathrm{n} \leq 0 \wedge \ldots \wedge \ldots \wedge \ldots.$$

To obtain an invariant map from these templates, we need to instantiate the set of parameters $\{\alpha, \alpha_{\mathrm{i}}, \alpha_{\mathrm{j}}, \alpha_{\mathrm{k}}, \alpha_{\mathrm{m}}, \alpha_{\mathrm{n}}, \beta, \beta_{\mathrm{i}}, \beta_{\mathrm{j}}, \beta_{\mathrm{k}}, \beta_{\mathrm{m}}, \beta_{\mathrm{n}}, \gamma, \gamma_{\mathrm{i}}, \gamma_{\mathrm{j}}, \gamma_{\mathrm{k}}, \gamma_{\mathrm{m}}, \gamma_{\mathrm{n}}\}$. We proceed by constructing a system of constraints, say Ψ, over the set of template parameters that

imposes the invariant conditions on the template map, following a classical approach from the literature [5, 28]. We omit the details for brevity. Unfortunately, even for this small example, we obtain a system of non-linear arithmetic constraints which exceeds the capacity of our constraint solver. Our idea is to scale the invariant generation engine by using information obtained from abstract interpretation as well as from concrete and symbolic runs of the program.

We first observe that for this example, some components of the required invariants can be generated by techniques based on abstract interpretation, e.g., by using octagon and polyhedral domains [7, 24]. By running INTERPROC (using PPL) on this example, we obtain the following invariant map η_α that annotates the loop locations with valid assertions:

$$\eta_\alpha.4 = \ \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0 \,, \qquad \eta_\alpha.5 = \ \mathtt{n} \geq \mathtt{j} \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0 \wedge \mathtt{j} \geq 0 \wedge \mathtt{n} \geq 1 \,,$$
$$\eta_\alpha.6 = \ \mathtt{n} + \mathtt{m} \geq \mathtt{k} \wedge \mathtt{n} \geq \mathtt{j} + 1 \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{k} \geq \mathtt{j} \wedge \mathtt{i} \geq 0 \wedge \mathtt{j} \geq 0.$$

While theoretically the analysis could have found all polyhedral relationships, in practice tools like INTERPROC employ several heuristics that sacrifice precision for speed. In this case, INTERPROC misses the inequality $\mathtt{n} + \mathtt{m} \geq \mathtt{i}$ valid at lines 5 and 6 and crucial for proving the assertion. Our algorithm takes the output generated by the abstract interpreter and uses it as an initial, static strengthening to support constraint based invariant generation.

In the second step, our algorithm collects dynamic information by executing the program. We first present a direct approach that uses program states to compute additional constraints that support invariant generation. Then, we show an extension that can handle unbounded collections of states. The extended method uses symbolic execution to collect such sets of states. We formalize these direct and symbolic approaches in Section 4.

Direct approach. Our direct approach starts with a collection of some reachable program states, which can be obtained by applying test generation techniques. We only track states at the head locations of the loops. Suppose we get the following set of states $\{s_1, \dots, s_4\}$ by running the program on test inputs:

$$s_1 = (pc = 4, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1), \quad s_2 = (pc = 4, \mathtt{j} = 3, \mathtt{i} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1),$$
$$s_3 = (pc = 5, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1), \quad s_4 = (pc = 6, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1).$$

Here, the variable pc represents the control location. We shall use these states to simplify the constraints for invariant generation.

We observe that since template expressions must be true for all reachable program states, in particular, they must hold for the states collected by testing. That is, for each reachable state we can substitute program variables appearing in the template by their values determined by the states and use this information to strengthen the constraint Ψ.

Thus, we can conjoin the following set of linear inequalities to the system of constraints Ψ, which determines the invariant map:

$$\begin{array}{ll} \alpha + \alpha_\mathtt{m} + \alpha_\mathtt{n} \leq 0 \,, \text{ from } s_1 & \alpha + 3\alpha_\mathtt{j} + \alpha_\mathtt{m} + \alpha_\mathtt{n} \leq 0 \,, \text{ from } s_2 \\ \beta + \beta_\mathtt{m} + \beta_\mathtt{n} \leq 0 \,, \text{ from } s_3 & \gamma + \gamma_\mathtt{m} + \gamma_\mathtt{n} \leq 0 \,, \text{ from } s_4 \end{array}$$

These additional constraints are linear. They can be applied by the solver to trigger a series of simplification steps. After the solving succeeds, we obtain the following invariant map:

$$\eta.4 = \ \mathtt{n} \leq \mathtt{m}, \mathtt{i} \geq 0 \,, \qquad\qquad \eta.5 = \ \ \mathtt{n} + \mathtt{m} \geq \mathtt{i}, \mathtt{n} \leq \mathtt{m}, \mathtt{i} \geq 0 \,,$$
$$\eta.6 = \ \mathtt{n} + \mathtt{m} \geq \mathtt{i}, \mathtt{k} \geq \mathtt{j}, \mathtt{n} \leq \mathtt{m}, \mathtt{i} \geq 0 \,.$$

Symbolic approach. We observe that we can simulate the effect of dynamic simplification using a large/unbounded set of reachable states. For this purpose we use symbolic execution, which computes assertions representing sets of reachable program states. We assume the example discussed so far and three reachable symbolic states below:

$$\varphi_1 = (pc = 4 \wedge \mathtt{i} = 0 \wedge \mathtt{n} \leq \mathtt{m}) \,,$$
$$\varphi_2 = (pc = 5 \wedge \mathtt{i} = 0 \wedge \mathtt{j} = 0 \wedge \mathtt{n} \geq 1 \wedge \mathtt{n} \leq \mathtt{m}) \,,$$
$$\varphi_3 = (pc = 6 \wedge \mathtt{i} = 0 \wedge \mathtt{j} = 0 \wedge \mathtt{k} = 0 \wedge \mathtt{n} \geq 1 \wedge \mathtt{n} \leq \mathtt{m}) \,.$$

These symbolic states can be applied to derive additional linear constraints on the template parameters. Due to the reachability of $\varphi_1, \varphi_2,$ and φ_3 the implications

$$\varphi_1 \rightarrow \eta.4 \,, \ \ \varphi_2 \rightarrow \eta.5 \,, \ \ \varphi_3 \rightarrow \eta.6$$

hold for all valuations of program variables. The validity of these implications can be translated into a linear constraint, say Φ, over template parameters. (See Section 4 for details.) We conjoin the constraint Φ with the constraint Ψ that encodes the invariance condition. As a result, the solver performs additional simplifications that lead to improved running time.

Relevant strengthening. In fact, after running our algorithm we can discover which inequalities computed using abstract interpretation and added as strengthening to the program were actually useful for finding the invariant that proves the assertion. This information is crucial for keeping minimal the number of facts reported to the software model checker as refinement predicates. For this purpose, we examine the solutions that the constraint solver assigned to the variables encoding the implication validity. For our example, the following inequalities found by INTERPROC were useful: $\mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0$ at line 4, $\mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0$ at line 5, and $\mathtt{k} \geq \mathtt{j} \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0$ at line 6.

3 Preliminaries

We start by describing the invariant-based approach for the verification of temporal safety properties and illustrate constraint-based invariant generation.

Programs and Invariants. We assume an abstract representation of programs by transition systems [23]. A *program* $P = (X, \mathcal{L}, \ell_{\mathcal{I}}, \mathcal{T}, \ell_{\mathcal{E}})$ consists of a set X of variables, a set $\mathcal{L}$ of control locations, an initial location $\ell_{\mathcal{I}} \in \mathcal{L}$, a set $\mathcal{T}$ of transitions, and an error location $\ell_{\mathcal{E}} \in \mathcal{L}$. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in \mathcal{L}$ are control locations, and ρ is a constraint over variables from $X \cup X'$. The variables from X denote values at control location ℓ, and the variables from X' denote the values of the

variables from X at control location ℓ'. The error location $\ell_{\mathcal{E}}$ is used to represent assertion statements. Each failed assertion leads to $\ell_{\mathcal{E}}$. We assume that the error location $\ell_{\mathcal{E}}$ does not have any outgoing transitions. The sets of locations and transitions naturally define a directed graph, called the *control-flow graph* (CFG) of the program, which puts the transition constraints at the edges of the graph.

A *state* of the program P is a valuation of the variables X. The set of all states is denoted by Σ. We shall represent sets and binary relations over states using constraints over X and X' in the standard way. A *computation* of P is a sequence of location and state pairs $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots$ such that ℓ_0 is the initial location and for each consecutive $\langle \ell_i, s_i \rangle$ and $\langle \ell_{i+1}, s_{i+1} \rangle$ there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A state s is *reachable* at location ℓ if $\langle \ell, s \rangle$ appears in some computation. The program is *safe* if the error location $\ell_{\mathcal{E}}$ does not appear in any computation. A *path* of the program P is a sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots$ of transitions, where ℓ_0 is the initial location. The path π is *feasible* if there is a computation $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots$ such that each consecutive pair of states (s_i, s_{i+1}) is induced by the corresponding transition, i.e., $(s_i, s_{i+1}) \models \rho_i$. A path that ends at the error location is called an *error path* (or *counterexample path*).

An *invariant* of P at a location $\ell \in \mathcal{L}$ is a super set of states that are reachable at ℓ, which we represent by an assertion over X. An *inductive invariant map* assigns an invariant to each program location such that for each transition $(\ell, \rho, \ell') \in \mathcal{T}$ the implication $\eta.\ell \wedge \rho \rightarrow (\eta.\ell')'$ is valid, where $(\eta.\ell')'$ is the assertion obtained by substituting variables X with the variables X' in $\eta.\ell'$. We observe that due to the invariance condition we have $\eta.\ell_{\mathcal{I}} = true$. An invariant map is *safe* if it assigns an empty set to the error location, i.e., $\eta.\ell_{\mathcal{E}} = false$.

A safe inductive invariant map serves as a proof that the error location cannot be reached on any program execution, and hence that the program is safe. The *invariant-synthesis* problem is to construct such a map for a given program.

Constraint-Based Invariant Generation. In the *constraint-based approach* [6, 20, 25, 26, 27] to invariant generation, the computation of an invariant map is reduced to a global constraint solving problem over the program locations. The approach consists of three steps. First, a *template* assertion that represents an invariant for each location is fixed in an *a priori* chosen language. A template assertion refers to the program variables X as well as a set of parameters. A parameter valuation determines an invariant. Second, a set of *constraints* over these parameters is defined in such a way that the constraints correspond to the definition of the invariant. This means that every solution to the constraint system yields a safe inductive invariant map. Third, a valuation of parameters is obtained by solving the resulting constraint system.

The language of arithmetic has been widely used to specify invariant templates [20, 25, 26]. A *linear inequality* over the variables $X = (x_1, \ldots, x_n)$ is an expression of the form $a_0 + a_1 x_1 + \ldots + a_n x_n \leq 0$ if $a_0, \ldots, a_n$ are rational numbers. The language of linear arithmetic consists of conjunctions of linear inequalities. An invariant template in linear arithmetic treats $\alpha_0, \ldots, \alpha_n$ as unknown parameters. For example, the template $\alpha + \alpha_x x + \alpha_y y + \alpha_z z \leq 0$ represents a linear inequality term over the variables x, y, and z. Here, the parameters are $\alpha, \alpha_x, \alpha_y$, and α_z. A possible template instantiation is $-4 + x + 2y - z \leq 0$.

An invariant template and its expressiveness are determined by the number of conjuncts that appear in the template for each program location. Adding more conjuncts increases the expressive power at the cost of a more expensive constraint solving task. Usually, templates are constructed incrementally, by starting with the weakest template that assigns a single conjunct to each program location and then refining it by adding additional conjuncts if the constraint solving fails to instantiate the template.

Given a template specification for an invariant map, we generate a set of constraints that encode the inductiveness and safety conditions. To encode the inductiveness condition, we generate a constraint $\eta.\ell \wedge \rho \rightarrow (\eta.\ell')'$ for each transition (ℓ, ρ, ℓ'). Note that this implication is implicitly universally quantified over X and X'. Furthermore, the conjunction of such implications for all transitions is existentially quantified over the template parameters. Using Farkas' lemma [28], we eliminate universal quantification. The result is a set of existentially quantified non-linear constraints over the template parameters as well as over the parameters introduced by Farkas' lemma (see [25] for the technical details). Techniques involving Gröbner bases and real quantifier elimination can be used similarly to generate and solve constraints for more general polynomial constraints [20, 26], and for the combined theory of linear arithmetic and uninterpreted functions [2].

We assume a function InvGenSystem that computes constraints from programs and templates. An application of InvGenSystem on a program and templates for each program location produces a constraint over the template parameters that encodes the invariant map conditions. For the implementation details see [2, 5].

We illustrate InvGenSystem using a single transition between location ℓ and ℓ' with the transition relation $x \leq y \wedge x' = x + 1 \wedge y' = y$. We assume a template $\varphi = (\alpha + \alpha_x x + \alpha_y y \leq 0 \wedge \beta + \beta_x x + \beta_y y \leq 0)$ consisting of two conjuncts at the location ℓ, and a singleton conjunction $\psi = (\gamma + \gamma_x x + \gamma_y y \leq 0)$ at the location ℓ'. The starting point is the implication $\varphi \wedge \rho \rightarrow \psi'$. To simplify the exposition, we first eliminate the primed program variables and obtain $\varphi \wedge x \leq y \rightarrow \psi[x + 1/x]$, which we present in the matrix form below.

$$\begin{pmatrix} \alpha_x & \alpha_y \\ \beta_x & \beta_y \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -\alpha \\ -\beta \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} \gamma_x + 1 & \gamma_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq -\gamma$$

Now, we apply Farkas' lemma to encode the validity of implication and obtain the following constraint:

$$\exists \lambda \geq 0. \; \lambda \begin{pmatrix} \alpha_x & \alpha_y \\ \beta_x & \beta_y \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} \gamma_x + 1 & \gamma_y \end{pmatrix} \wedge \lambda \begin{pmatrix} -\alpha \\ -\beta \\ 0 \end{pmatrix} \leq -\gamma$$

This constraint determines the values of template parameters and the additional parameter λ. It contains non-linear terms that result from the multiplication of λ with $(\alpha_x \; \beta_x)$ and $(\alpha_y \; \beta_y)$.

Constraint Solving. The constraints generated above are non-linear, since they contain multiplication terms over the parameters from the invariant templates, as well as the additional parameters introduced by Farkas' lemma. The existing solving approaches include symbolic techniques based on instantiations and case splitting, e.g. [5], and using SAT solvers by applying an appropriate propositional encoding, e.g. [14].

For the rest of the paper, we assume a function Solve that takes as input a set of non-linear constraints and returns either a satisfying assignment to the constraints, or that the constraint set is unsatisfiable. Unfortunately, in all but the most basic programs, constraint-based invariant synthesis using the above technique is too expensive. For most realistic programs, the procedure Solve times out.

4   Constraint Simplification

We now describe how we can use additional static and dynamic information to restrict the search space determined by the set of static constraints. Technically, we do this by computing additional constraints on the program transition relation and on the template parameters and conjoining them with the constraint system defining invariant map. Program computations provide a source of such additional dynamic constraints.

INVGEN+ABSINT: Simplification from Abstract Interpretation. Our first simplification uses an abstract interpreter to compute program invariants, and uses the result of the abstract interpretation algorithm to strengthen the program transition relation. That is, suppose that η_α is an invariant map computed by an abstract interpretation algorithm. In our constraint generation, we replace the constraint $\eta.\ell \wedge \rho \rightarrow (\eta.\ell')'$ for a transition (ℓ, ρ, ℓ') with the constraint $\eta.\ell \wedge (\eta_\alpha.\ell \wedge \rho) \rightarrow (\eta.\ell')'$.

INVGEN+TEST: Simplification from Tests. Individual program computations can be used to simplify the constraints for invariant generation. The crux of the algorithm IN-VGEN+TEST lies in the observation that an invariant template must hold when partially evaluated on a reachable state of the program.

Let $t(X)$ be a template over the program variables X and s be a reachable program state. We write $t(s/X)$ to denote a template expression that is obtained from t by substituting each variable $x \in X$ with its value $s(x)$ in the state s. Then, the constraint $t[s/X]$ imposes an additional constraint over the template parameters. Note that this constraint is *linear*, i.e., its processing does not require application of expensive non-linear solving techniques.

We show the algorithm INVGEN+TEST in Figure 2. The algorithm takes as input a program P and an invariant template map η with parameters $\mathcal{P}$. It can return an invariant map for P, output that no invariant map exists for the given invariant templates, or find a counterexample to the program safety. There are three conceptual steps of the algorithm. The first step (line 1) constructs a set Ψ of constraints on the invariant template parameters that encode the initiation, inductiveness, and safety conditions. The second step (lines 2–9) runs a set of tests and generates additional constraints on the parameters based on the test executions. Finally, the third step (line 10) solves the conjunction of the static constraints from line 1 and the additional constraints generated during testing.

The loop in lines 3–9 executes the program on a set of tests. We instrument the program so that for each program location ℓ reached in the test, the concrete values of all the program variables that appear in the template $\eta.\ell$ are recorded. If a test hits the error location, then of course, we have found a bug, and we return this error (lines 5,6). Otherwise, the recorded values provide an additional constraint on the template parameters.

```
            input
               P: program; η: invariant template map with parameters 𝒫
            vars
               Ψ : static constraint; Φ : dynamic constraint
            begin
    1          Ψ := InvGenSystem(P, η)
    2          Φ := true
    3          repeat
    4             s_1, ..., s_n := GenerateAndRunTest(P)
    5             if s_n(pc) = ℓ_ℰ then
    6                return "counterexample s_1, ..., s_n"
    7             else
    8                Φ := Φ ∧ ⋀_{i=1}^{n} (η.s_i(pc))[s_i/X]
    9          until no more tests
    10         if  𝒫* := Solve(Ψ, Φ) succeeds then
    11            return "inductive invariant map η[𝒫*/𝒫]"
    12         else
    13            return "no invariant map for given template"
            end.
```

Fig. 2. Algorithm INVGEN+TEST for invariant generation supported by dynamic simplification using program executions. InvGenSystem creates a constraint over the template parameters that encodes invariant map conditions for the program P, see Section 3. The function GenerateAndRunTest selects program computations.

For example, if the template for a location is $\alpha x + \beta y + \gamma \leq 0$, and a dynamic execution reaches this location with the concrete state $x = 35, y = -9$, we know that the parameters α, β, and γ must satisfy the constraint $35\alpha - 9\beta + \gamma \leq 0$. We call this a *dynamic constraint* on the parameters and add this constraint to the auxiliary constraint Φ.

The testing loop terminates due to an externally supplied coverage criterion. At this point, the constraint solver is invoked to find a satisfying assignment for the parameters in $\mathcal{P}$ that satisfy both the static constraints in Ψ and the dynamic constraints in Φ. If there is no such solution, the algorithm returns that there is no invariant map for the program using the current template map. On the other hand, any satisfying assignment provides an invariant map. Our algorithm maintains the invariant that at any point in lines 3–13, a satisfying assignment to the constraints $\Psi \wedge \Phi$ is guaranteed to be a valid invariant map.

INVGEN+SYMB: Simplification from Symbolic Execution. We observe that the basic algorithm conjoins dynamic, *linear* constraints for each state that is reached by the test generator. A large number of such constraints may overwhelm the constraint solver, despite their low processing cost. We improve the basic algorithm by taking into account *sets* of reachable states using a single strengthening constraint.

We assume a template $t(X)$ and a set of reachable states represented by an assertion $\varphi(X)$. We can obtain such sets of states by performing symbolic execution along a collection of program paths. Then, the implication $\varphi(X) \rightarrow t(X)$ must hold for all valuations of X since every state in φ is reachable.

3	**repeat**
4.1	$\pi := \mathsf{GeneratePath}(P)$
4.2	$(* \; \pi_i = (\ell_i, \rho_i, \ell_{i+1}) \text{ for } 1 \leq i \leq n \; *)$
5	**if** $\ell_{n+1} = \ell_{\mathcal{E}}$ and π is feasible **then**
6	**return** "counterexample π"
7	**else**
8.1	$\varphi := (\exists X. \; \rho_1 \circ \cdots \circ \rho_n)[X/X']$
8.2	$\Phi := \Phi \wedge \; \mathsf{Encode}(\varphi \rightarrow \eta.\ell_{n+1})$
9	**until** no more paths

Fig. 3. Algorithm INVGEN+SYMB. It can be obtained by replacing lines 3–9 of the algorithm INVGEN+TEST with the above statements. The function GeneratePath selects program paths. Encode creates *linear* constraints over template parameters that encode the validity of the given implication.

Following the method in Section 3, we encode the validity of the implication by a constraint over the template parameters. In this case, the encoding yields *linear* constraints. In contrast to the cases when the left-hand side of the implication contains template assertions, in the above implication program variables have *constant* coefficients. Thus, when multiplying additional parameters (appearing due to the application of Farkas' lemma) with coefficients attached to the program variables we obtain linear terms, which, in turn, result in linear constraints.

For example, we consider a template $t(x, y, z)$ that consists of two conjuncts $\alpha + \alpha_x x + \alpha_y y + \alpha_z z \leq 0 \; \wedge \; \beta + \beta_x x + \beta_y y + \beta_z z \leq 0$. We assume a set of states $\varphi = (-x \leq 0 \; \wedge \; -y \leq 0 \; \wedge \; x + y - z \leq 0)$ reached by symbolic execution. The encoding of the implication $\varphi \rightarrow t$ yields the constraint

$$\exists \Lambda \geq 0. \; \Lambda \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{pmatrix} = \begin{pmatrix} \alpha_x & \alpha_y & \alpha_z \\ \beta_x & \beta_y & \beta_z \end{pmatrix} \wedge \Lambda \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} -\alpha \\ -\beta \end{pmatrix} \; ,$$

which is clearly linear.

We assume a function Encode that translates an implication between an assertion representing a set of states and a template into a linear constraint over template parameters. Our extended algorithm INVGEN+SYMB applies Encode on sets of reachable states computed by symbolic execution of the program. The algorithm is presented in Figure 3. Since it extends the basic algorithm INVGEN+TEST by adding the symbolic treatment of reachable states, we only present the modified part.

The algorithm INVGEN+SYMB interleaves symbolic execution and collection of constraints. It relies on an external function GeneratePath that selects paths through the control-flow graph of the program, see line 4.1. For a given path, we compute an assertion representing states that are reachable by executing its transitions, see line 8.1. We use the relational composition operator $\circ$, which is defined by $\rho \circ \rho' = \exists X''. \; \rho[X''/X'] \wedge \rho'[X''/X]$, to compute the transition relation of the whole path. The existential quantification in line 8.1 projects this relation to the successor states φ, i.e., it computes the range of the relation. We use variable renaming to keep the resulting assertion consistent with the templates over program variables. We conjoin the constraint resulting from the translation of the implication between the reachable states

Table 2. Comparison of variations of invariant verification techniques and INTERPROC on additional benchmark problems inspired by [21]. "✓" and "×" indicate whether the invariant computed by INTERPROC proves the assertions, and "T/O" stands for time out.

File	INTERPROC	INVGEN	INVGEN+Z3	INVGEN + INTERPROC	INVGEN+Z3 + INTERPROC + SYMB	INVGEN + INTERPROC + SYMB
Seq	×	23.0s	1s	**0.5s**	6s	**0.5s**
Seq-z3	×	23.0s	9s	**0.5s**	6s	**0.5s**
Seq-len	×	T/O	T/O	T/O	4s	**2.8s**
nested	×	T/O	T/O	17.0s	3s	**2.3s**
svd(light)	×	T/O	T/O	**10.6s**	T/O	14.2s
heapsort	×	T/O	T/O	19.2s	48s	**13.3s**
mergesort	×	T/O	**52s**	142s	T/O	170s
SpamAssassin-loop	✓	T/O	5s	**0.28s**	1s	0.4s
apache-get-tag	×	**0.4s**	10s	0.6s	3s	0.7s
sendmail-fromqp	×	0.3s	5s	0.3s	5s	**0.3s**
Example1(b)	×	T/O	T/O	0.4s	1s	0.35s

φ and the corresponding template $\eta.\ell_{n+1}$ to the dynamic constraint Φ before proceeding with the next path. We assume an external procedure that selects a finite set of paths. In our implementation, we apply directed symbolic execution that attempts to unroll loops at least one time.

The following theorem states that our optimizations are sound (and relatively complete).

Theorem 1. *[Correctness] If Algorithm* INVGEN+ABSINT, INVGEN+TEST, *or* IN-VGEN+SYMB *on input program P and invariant template map η returns (a) "counterexample $s_1, \ldots, s_n$," then there is an execution of the program that reaches the error location; (b) "inductive invariant map η^*," then η^* is an invariant map for P, and the program P is safe; (c) "no invariants with template η," then there is no invariant map for P with the given invariant template map η.*

5   Experiences

Implementation. We implemented the algorithms INVGEN+TEST and INVGEN+SYMB using SICStus Prolog [29], the linear arithmetic solver clp(q,r) [18] and the Z3 solver [8] as the backend to solve non-linear constraints. When describing the application of INVGEN together with Z3, we shall write INVGEN+Z3. We apply the IN-TERPROC [22] tool for abstract interpretation over numeric domains, and use the PPL backend for polyhedra, mainly due to its source code availability. In principle, a variety of other tools could be used instead, e.g., the ASPIC tool implementing the lookahead widening and acceleration techniques [11, 12]. INVGEN provides a frontend for C programs, which relies on CIL infrastructure for C program analysis and transformation and abstracts from non-arithmetic operations appearing in the input program. We implement the following additional variable elimination optimization. The additional constraints obtained from dynamic and static strengthening are linear. In particular, the additional variables that encode implication between symbolic states and templates, Λ

in the previous section, can be eliminated. We perform this simplification step before applying the (expensive) techniques for solving non-linear constraints. For our constraint logic programming-based implementation, this results in a reduction of the number of calls to the linear arithmetic solver. When using the SAT approach, it allows us to avoid applying the propositional search to constraints that can be solved symbolically.

In our experimental evaluation, we observed that INVGEN+TEST and INVGEN+SYMB offer similar efficiency improvement, with a few exceptions when INVGEN+SYMB was significantly better. To keep the tables with experimental data compact, we only describe evaluation of the strengthening that uses symbolic execution INVGEN+SYMB.

Software Verification Challenge Benchmarks. We applied INVGEN on a suite of software verification challenge programs described in [21]. The examples in this benchmark are extracted from large applications by mining a security vulnerability database for buffer overflow problems. We use the corrected versions of these programs, using the buffer access checks as assertions. The suite consists of 12 programs.[1] Using polyhedral abstract domain, INTERPROC computes invariants that are strong enough to prove the assertion for half of them. The constraint based invariant generation together with the SAT-based encoding, i.e., INVGEN+Z3, generates invariants for all programs within 36.5 seconds of total time. Using the CLP backend, INVGEN handles 11 examples within 6.3 seconds, and times out on one program, which is handled by INVGEN+Z3 in 5 seconds. Using the static and dynamic strengthening described in this paper, we obtain the following running times. The combination INVGEN+Z3+INTERPROC+SYMB solves all examples in 29.5 seconds, while INVGEN+INTERPROC+SYMB handles all examples within 9.6 seconds. These experiments demonstrate that the various optimizations can have an effect on verification, but the running times were too short to draw meaningful conclusions.

Impact of Dynamic Strengthening. The collection from [21] did not allow us to perform a detailed benchmarking of our algorithm, since the running times on these examples were too short. We obtained a set of more difficult benchmarks inspired by [21] by adding additional loops and branching statements, and provide a detailed comparison that describes the impact of static and dynamic strengthening in isolation in Table 2. INTERPROC computes 50 inequalities for each loop head, which results in a significant increase in the number of variables in the constraint system. While being an obstacle for the propositional search procedure in Z3, the increased number of variables does not significantly affect the CLP-based backend since the additional variables appear in linear terms. In summary, the performance of INVGEN+Z3 decreases and the performance of INVGEN goes up by adding facts from INTERPROC.

Integration with BLAST. We have modified the abstraction refinement procedure of the BLAST software model checker [15] by adding predicate discovery using path invariants [3]. Table 3 shows how constraint based invariant generation can be effective for

---

[1] Due to short running times, we present the aggregated data and do not provide any table containing entries for individual programs.

refining abstractions. The number of counterexample refinement iterations required is reduced in all examples.

For several examples we achieved termination of previously diverging abstraction refinement, and for others the reduction ranges between 25 and 400 percent.

Summary. Our experimental evaluation leads to the following observations:

- For complex constraint solving problems, the additional strengthening facilitates significant improvement. It ranges from reducing the running time by two orders of magnitude to making timing out examples solvable within seconds.
- If the constraint solving is already fast in the purely static case, then the strengthening does not cause any significant running time penalty.

Table 3. INVGEN + INTERPROC + SYMB for predicate discovery in BLAST. We show the number of refinement steps required to prove the property.

File	BLAST	BLAST + INVGEN + INTERPROC + SYMB
Seq	diverge	8
Seq-len	diverge	9
fregtest	diverge	3
sendmail-fromqp	diverge	10
svd(light)	144	43
Spamassassin-loop	51	24
apache-escape	26	20
apache-get-tag	23	15
sendmail-close-angle	19	15
sendmail-7to8	16	13

References

1. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL, pp. 1–3. ACM Press, New York (2002)
2. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
3. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Proc. PLDI, pp. 300–309. ACM Press, New York (2007)
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: Proc. PLDI, pp. 196–207. ACM Press, New York (2003)
5. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using nonlinear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
6. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, pp. 84–96. ACM Press, New York (1978)

8. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Software Eng. 27(2), 1–25 (2001)

10. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, pp. 19–32. AMS (1967)

11. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in Linear Relation Analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)

12. Gopan, D., Reps, T.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)

13. 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)

14. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, pp. 281–292. ACM Press, New York (2008)

15. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: POPL 2004: Principles of Programming Languages, pp. 232–244. ACM Press, New York (2004)

16. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM Press, New York (2002)

17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12, 576–580 (1969)

18. Holzbaur, C.: OFAI clp(q,r) Manual, edn. 1.3.3. Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09 (1995)

19. Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 137–151. Springer, Heidelberg (2006)

20. Kapur, D.: Automatically generating loop invariants using quantifier elimination. Technical Report 05431 (Deduction and Applications), IBFI Schloss Dagstuhl (2006)

21. Ku, K., Hart, T., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: Proc. ASE (2007)

22. Lalire, G., Argoud, M., Jeannet, B.: The interproc analyzer, http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html

23. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Heidelberg (1995)

24. Miné, A.: The octagon abstract domain. Higher-Order and Symb. Comp. 19, 31–100 (2006)

25. Sankaranarayanan, S., Sipma, H., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)

26. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: Proc. POPL, pp. 318–329. ACM, New York (2004)

27. Sankaranarayanan, S., Sipma, H., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)

28. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)

29. The Intelligent Systems Laboratory. SICStus Prolog User's Manual. Swedish Institute of Computer Science, Release 3.8.7 (2001)

Test Input Generation for Programs with Pointers

Dries Vanoverberghe[1,*], Nikolai Tillmann[2], and Frank Piessens[1]

[1] Katholieke Universiteit Leuven, Belgium
{Dries.Vanoverberghe,Frank.Piessens}@cs.kuleuven.be
[2] Microsoft Research, Redmond, USA
nikolait@microsoft.com

Abstract. Software testing is an essential process to improve software quality in practice. Researchers have proposed several techniques to automate parts of this process. In particular, symbolic execution can be used to automatically generate a set of test inputs that achieves high code coverage.

However, most state-of-the-art symbolic execution approaches cannot directly handle programs whose inputs are pointers, as is often the case for C programs. Automatically generating test inputs for pointer manipulating code such as a linked list or balanced tree implementation remains a challenge. Eagerly enumerating all possible heap shapes forfeits the advantages of symbolic execution. Alternatively, for a tester, writing assumptions to express the disjointness of memory regions addressed by input pointers is a tedious and labor-intensive task.

This paper proposes a novel solution for this problem: by exploiting type information, disjointness constraints that characterize permissible configurations of typed pointers in byte-addressable memory can be automatically generated. As a result, the constraint solver can automatically generate relevant heap shapes for the program under test. We report on our experience with an implementation of this approach in Pex, a dynamic symbolic execution framework for .NET. We examine two different symbolic representations for typed memory, and we discuss the impact of various optimizations.

Keywords: Test input generation, symbolic execution, pointers.

1 Introduction

Today, testing is still by far the most effective way to improve software quality. Recently, there is a lot of effort to automate different parts of the testing process. One aspect is test input generation for an open program, i.e. a program that takes

---

inputs. The challenge is to generate a small set of test inputs that maximizes code coverage.

Symbolic execution[1] is a well known static analysis technique to generate test inputs, where the program is executed with symbolic inputs instead of concrete inputs. A constraint solver is used to compute test inputs for particular execution paths. Since its early introduction, there has been a tremendous increase in computing power which paved the road for engineering more efficient constraint solvers and more precise analysis tools. Not surprisingly, symbolic execution for test input generation has become popular lately (e.g. Java Pathfinder [2], EXE [3], Cute [4], Sage [5], Pex [6], ...). Most of these tools use a variant of symbolic execution where the program is repeatedly executed with concrete inputs. During a concrete execution, the program is monitored to build a symbolic representation of the executed path; the next test inputs are constructed in such a way that they will exercise a new execution path. Constraints that are outside of the scope of the employed constraint solver can be simplified by using observed concrete values instead of symbolic values. This variant has been called DART [7], concolic execution [4], and dynamic symbolic execution[5,6]. The many tools implementing it show its relevance in practice (EXE [3], Cute [4], Sage [5], Pex [6], Yogi [8], Vigilante [9], Bitscope, ...).

In the context of static analysis tools, reasoning about programs with pointers has traditionally been challenging. For test input generation this is not different: Some tools based on dynamic symbolic execution don't support symbolic pointer reasoning and use the concrete value for pointers as an under-approximation instead. For instance, EXE [3] uses concrete values whenever it encounters a double dereference and Sage [5] always uses concrete values. Although this means that the exploration can sometimes be incomplete, for testing tools this is appropriate. If symbolic pointers are supported, simplifying assumptions often make pointer reasoning incomplete (for example, treating pointers as references instead of integers with arbitrary arithmetic [4]).

In this work we focus on test input generation for programs manipulating (complex) data structures with pointers. A common approach is to first assume that an invariant holds for the data structure; then to apply an operation on the data structure; and finally to assert that the invariant still holds. Expressing the invariant for arbitrary pointers is often tedious, as a large portion of the invariant usually states that different pointers point to different memory regions which do not overlap in type-incorrect ways. This is a property that is guaranteed by safe managed languages such as Java and C#, but it is not guaranteed by languages that allow unsafe memory accesses, such as C and C++.

We studied the problem in the context of Pex [6], a dynamic symbolic execution tool for the Common Intermediate Language (CIL) of the .NET Framework. CIL consists of a strongly typed verifiable core, extended with unsafe instructions (e.g. for unsafe memory accesses through pointers) that are available only when the program is sufficiently trusted by the user. By using these unsafe instructions, C programs can be translated to CIL.

The contributions of this paper are the following:

- We propose two different symbolic representations for unsafe memory. The main idea of these representations is to exploit the type information that is present in the CIL to the fullest extent possible.
- We present a number of axioms that characterize that pointers only overlap in type-correct ways.
- We introduce a two-phase solving scheme to improve the efficiency of the solving process by performing multiple incremental queries to the solver.
- We report on an experimental evaluation based on an implementation in Pex.

The remainder of this paper is structured as follows: Section 2 explains more details about dynamic symbolic execution. Section 3 motivates why automatic test input generation for programs with pointers is challenging. Next, we shortly introduce our type system (Section 4) before explaining how we model memory (Section 5) and how we enforce disjointness of input data accessed through typed pointers (Section 6). In Section 7 we discuss how we can improve the performance of the resulting system. Section 8 contains an evaluation of our approach using red-black trees and linked lists as data structure. Finally, we treat related work in Section 9 and conclude.

2   Background: Dynamic Symbolic Execution

Symbolic execution [1] is a technique to explore the behavior of a program under all possible inputs. Instead of using concrete inputs, the program is executed with symbols representing arbitrary values. As a result, the inputs are partitioned into equivalence classes that follow the same execution path. This path is represented by the path-condition, a conjunction of constraints on the symbolic inputs. At each branch during the execution of the program, the inputs are split into two equivalence classes by conjoining the (negated) branch condition with the path-condition. A satisfiability modulo theory (SMT) solver is used to check whether the resulting path-conditions are feasible and to compute a set of inputs that represents that particular execution path.

Theoretically, given a correct (sound and complete) constraint solver and a correct symbolic execution engine, symbolic execution of programs with a finite number of finite paths is equivalent to program verification. In reality, the symbolic execution engine will often be limited in its completeness due to instructions with complex behavior such as floating point operations or code that is outside the scope of the symbolic execution engine such as operating system calls.

Dynamic symbolic execution (also named DART [7] or concolic execution [4]) is a variant of symbolic execution in which the symbolic constraints are gathered by monitoring the program during a concrete execution and maintaining a symbolic representation on the side. First the program is executed with arbitrary inputs. Then, a constraint solver is used to find inputs that drive the

```
Set J := false                              intuitively, J is the set of already...
loop                                        ...analyzed program inputs
    Choose program input i such that ¬J(i)  stop if no such i can be found
    Output i
    Execute P(i); record path condition C   in particular, C(i) holds
    Set J := J ∨ C
end loop
```

Algorithm 1. Dynamic symbolic execution

execution along new execution paths. The advantage of this approach is that even in the context of an imprecise representation, there will never be false positives. When the concrete execution diverges from the intended execution path it can be detected and reported. Furthermore, when precise symbolic execution is impossible, concrete values can be used to simplify the constraints. In that case, the set of covered execution paths is an underapproximation of the feasible execution paths, which is appropriate for testing.

Algorithm 1 shows the general dynamic symbolic execution algorithm.

For symbolic execution, the constraint solver needs to support a variety of different theories, such as bit-vectors with all common integer arithmetic operations to model machine integers, the theory of arrays with read and write-functions for arrays, and tuples to represent structs. Often, uninterpreted functions are used (e.g. for typing constraints); their possible interpretations are restricted, or fully determined, by introducing background axioms (potentially with quantifiers). Satisfiability modulo theory (SMT) checkers are efficient to reason about such a combination of theories. They handle quantified background axioms by instantiating them when a designated pattern occurs in the formula that is checked for satisfiability. This approach is only complete when the patterns are carefully chosen. Since modern SMT checkers can generate models, i.e. satisfying assignments, they can be used as constraint solvers for symbolic execution. In this work, we use Z3 [10] as the constraint solver.

3   Motivating Example

In the following, we will illustrate the problem of test input generation for programs whose inputs are pointers. Consider the program in Figure 1. The function *test* tests the method *enqueueTail*, a part of the enqueue operation of linked lists. It takes the tail pointer p, a pointer to a freshly initialized node q and a new value *val* as input and adds the value as last element of the list. In the beginning of the test, we assume that p and q are not null and different. Furthermore, the *next* field of both p and q must be null. After the *enqueueTail* operation, the value of the next node of p must equal the new value.

To cover this program, a dynamic test generation tool must generate values for pointers, i.e. memory addresses, and assign values to the memory locations at these addresses. In the first execution, arbitrary values can be assigned to

```
void enqueue(Queue* q, int val) {
    Node* newNode = malloc(sizeof(Node));
    enqueueTail(q -> tail, newNode, val);
    q -> tail = newNode; }
void enqueueTail(Node* p, Node* q, int val) {
    q -> value = val;    p -> next = q; }
void test(Node* p, Node* q, int val) {
    Assume(p != 0 && q != 0 && p != q);
    Assume(p -> next == 0 && q -> next == 0);
    enqueueTail(p, q, val);
    Assert(p -> next -> value == val); }
```

```
struct Node {
    int value;
    Node* next;
}
struct Queue {
    Node* head;
    Node* tail;
}
```

Fig. 1. A motivating example based on linked lists

the inputs; we choose the value 0 for pointers, and 0 for integers. During the execution, a symbolic representation is maintained, in which the pointers are treated as regular integers. (We use this representation of pointers as it reflects the encoding of pointer operations at the level of the execution machine, where all high-level type information has been erased.) In the first execution, p is null, so the path-condition will be $p == 0$. To explore new behavior, we need to find a value for p such that $p! = 0$. The constraints will be solved incrementally, and the final constraint to pass the first assumption is $p != 0 \&\& q != 0 \&\& p != q$. The constraint solver might find a solution such as $p = 1, q = 2$, where 1 and 2 are integers that represent the addresses of the second and third byte of the addressable memory. During the next execution, this will likely result in an access violation when trying to read the value of the next field of p, since the program does not have access to these locations when it is executed within a regular process of a typical operating system. Of course, this technical issue can be solved by allocating a big chunk of memory before any test is executed, and adding additional constraints on p and q to express that these pointers must be in that memory region.

Next, in order to pass the second assumption, we again send a set of constraints to the constraint solver. This time, the solver will not only give a model for the inputs, but also for some values in memory. Our tool parses this model and initializes the memory with the supplied values before executing the test. For example, in this case it would assign null to the *next* fields of p and q. Although this is not really challenging, integration testing tools (like Sage [5] and EXE [3]) often don't support this. For unit testing tools, dealing with pointers as input is essential.

Besides these technical issues, the dynamic symbolic execution process will report an assertion violation. This is because the program fails to specify the assumption that the memory regions, to which p and q point, do not overlap (are disjoint). Indeed, if for instance $q = p + 4$ then the next field of p has the same address as the value field of q. Since the next field of p is updated after the value of q is set to *val*, the value of q is destroyed. Therefore, the value of the next node of p is not equal to *val*.

In practice, it is surprisingly tedious to express all disjointness assumptions. Especially for complicated data structures, like trees or graphs, where the number of disjointness constraints is quadratic in the number of nodes. Forgetting one assumption can lead to a test case that is particularly hard to debug. Furthermore, our experience shows that developers are likely to forget such assumptions as they are often taken for granted.

In this paper, we seek to exploit the information present in the type system, and the signature of the test function. We restrict input generation for pointers in such a way that pointers are always used in a type-correct way. As a result, it is no longer necessary to encode this quadratic amount of disjointness constraints manually. Instead, the type information is encoded in the constraints for the constraint solver.

4   Types

In this section, we give a short sketch of the type system that we considered – a small subset of the type system of full CIL that we found relevant for unsafe memory operations. Figure 2 shows the syntax of types in our system. A type can either be a primitive type or a struct type. $I1$ is a byte, $I2$ a two-byte entity, and so on. $R4$ represents a four-byte float, and so on. Struct types are essentially a list of types with labels. In this way, types are basically trees where the leaves are primitive types and the nodes are struct types. Depending on the architecture of the machine, pointers would be represented as $I4$, or $I8$. We will use $I8$ in the following.

$$Type \; := \; PrimitiveType \; | \; StructType$$
$$PrimitiveType \; := \; I1 \, | \, I2 \, | \, I4 \, | \, I8 \, | \, R4 \, | \, R8 \, | \, \ldots$$
$$StructType \; := \; Type \; Label; \; StructType \, | \, Type \; Label$$

Fig. 2. Syntax types

We do not consider empty structs. We assume the presence of the function $sizeof$ that returns the size of a type in memory in bytes, and the function $nextOffset$ that gives the offset of the second label of a struct in bytes. The label represents a named field of a struct. The functions $nestedIn$ and $unnested$ can be used to test whether one type is nested in another type or not.

5   Memory Representations

This section describes how memory is modeled to support precise test input generation for unsafe pointers. From the point of view of the concrete execution engine, memory is just one big byte array. Whenever a value at a particular offset is read, the execution engine reads a number of consecutive bytes from memory and converts them to a value of the requested type.

The most precise way to model memory symbolically is to stay as close to the concrete semantics as possible. Logically, pointers can be encoded as regular integers, and memory can be represented as a map from integers to bytes. Reads and writes are represented as selecting from this map or updating it at a number of consecutive indices. This precision comes with a cost for every memory access with a type bigger than a byte. To reduce this cost, we exploit the typing information to simplify the memory representation. Since well-typed pointers can not overlap, we can split the memory into different maps according to the type. We explored two different variants of this scheme:

map per type. For each type T, we keep a map MM_T from integers to values of type T. For struct types, this means that the value of the field can be retrieved in two ways. For example, assume we have a struct T with a field of type S: First, we can select the entire struct of type T from the MM_T and get the value of its field. Alternatively, we can compute the address of the field and select the value directly from MM_S.

To keep the different maps consistent, we introduce an axiom over the maps that relates the initial memory maps. Whenever we write a complete struct to a pointer, we update both the memory map of the struct and the memory maps of its fields recursively. Furthermore, we use typing information that is present in CIL to reverse engineer when an assignment to a pointer is an assignment to a field of a struct. In that case, we do not only update the field, but also the struct. Unfortunately, when complex pointer manipulation operations are performed, it might not always be statically known that a pointer points to a field of a struct. In this case, the symbolic representation is imprecise with respect to the real representation.

map per primitive type. We only maintain maps for primitive types. Reading or writing complete structs is done recursively over all fields. This representation is no longer imprecise since the constraint solver now reasons about the relation between different pointers.

6   Enforcing Disjointness

As mentioned before, the memory representations introduced in Section 5 are only precise under the assumption that pointers separate memory into disjoints memory regions that are only accessed according to one particular type (or compatible types in the case of nested structs). In this section we explain how we use the typing information to enforce this assumption.

To encode the typing information for the constraint solver, we want to express that a pointer p has type T (e.g. $typeOf(p) == T$). Because our type system contains structs, it is possible though that one pointer has multiple types. Consider a pointer to a struct T whose first field has type S. This pointer has both T and S as type. To this end, we could introduce a relation $typed(T, p)$ to express that a pointer p has type T.

However, using such a relation as a basic block of our definitions would be inefficient: We would have to create constraints to forbid all illegal combinations

of types, e.g. to indicate that a $typed(byte, p)$ and $typed(int, p)$ is mutually exclusive. To achieve better performance, we stratify types according to their type level Essentially, the type level is the largest nesting depth. If we conceptually think of a type as a tree, the type level is the height of the tree.

For each type level, we define an uninterpreted function $typeOf_{typeLevel}$ that takes a pointer as input and returns a value representing the type of the pointer. Now we can define $typed(T, p)$ as syntactic sugar for $typeOf_{typeLevel(T)}(p) == typeConstant(T)$. Since the $typeOf$ symbols are functions, the theory of equality over uninterpreted function symbols can infer that $typeOf_l(p)! = typeOf_l(q)$ implies that p and q are different. Using only the predicate $typed$, these implications would have to be encoded as quantifiers, which is potentially less efficient.

The semantics of these uninterpreted functions is given by the axioms defined in Figure 3. Whenever a pointer to a struct type is typed, then the pointers to the fields of this struct are also typed according to their type. Furthermore, pointers that are typed must always be in a predefined region of memory that we allocated for this purpose. The constants $vmbase$ and $vmsize$ represent the base address and size of this region.

$\forall (T : Type, \ t : label, \ S : StructType, \ p : I8), \ \underline{typed((T \ t; S), \ p)}$
$\quad => \ typed(T, \ p) \ \&\& \ typed(S, \ (p + nextOffset(T \ t; S)))$
$\forall (T : Type, \ t : label, \ p : I8), \ \underline{typed((T \ t), \ p)} \ => \ typed(T, \ p)$
$\forall (T : Type, \ p : I8), \ \underline{typed(T, \ p)}$
$\quad => \ p \geq vmbase \ \&\& \ p + sizeof(T) \leq vmbase + vmsize \ \&\& \ p + sizeof(T) > p$

Fig. 3. Axioms over $typed$ function

Figure 4 shows the disjointness axioms that apply to pointers and Figure 5 introduces a number of helper functions.

First, two pointers with same type must either be equal or they do not overlap. Second, two pointers with different types where neither of the types is nested inside the other type never overlap. Finally, when one type is nested in another type, a pointer to the nested type can either be correctly embedded with respect to the pointer of the other type or both pointers do not overlap. Correctly embedded means that if the nested type is equal to a field type, then the embedded pointer can be equal to this pointer. Alternatively, if the nested type is nested inside a field type, then the embedded pointer can be embedded in the pointer to the field. Together with the axiom to propagate type information to the fields of a struct, these three axioms precisely define how pointers can relate to each other.

In Section 2, we mentioned that SMT solvers need a carefully designed (set of) patterns to instantiate quantifiers. In Figure 3 and 4, these patterns are illustrated by underlining them. We do not provide patterns for the first quantifier in the last two disjointness axioms because they are statically expanded. Two patterns in one quantifier represent one multi-pattern rather than two patterns. The patterns in our axioms only occur on the left hand side of an implication, and the right hand side will only generate the same pattern with terms that are structurally smaller. This corresponds to defining a function by recursion

$\forall \ (T : Type, p1 : I8, \ p2 : I8), \ \underline{typed(T, \ p1)} \ \&\& \ \underline{typed(T, \ p2)} \ =>$
    $p1 == p2 \parallel noOverlap(T, \ T, \ p1, \ p2)$
$\forall \ (T1 : Type, \ T2 : Type), \ unnested(T1, \ T2) \ =>$
        $\forall \ (p1 : I8, \ p2 : I8), \ \underline{typed(T1, \ p1)} \ \&\& \ \underline{typed(T2, \ p2)} \ =>$
        $noOverlap(T1, \ T2, \ \overline{p1, \ p2})$
$\forall \ (T1 : Type, \ T2 : Type), \ nestedIn(T1, \ T2) \ =>$
        $\forall \ (p1 : I8, \ p2 : I8), \ \underline{typed(T1, \ p1)} \ \&\& \ \underline{typed(T2, \ p2)} \ =>$
        $noOverlap(T1, \ T2, \ \overline{p1, \ p2}) \parallel correctlyEmbedded(T1, \ T2, \ p1, \ p2)$

Fig. 4. Disjointness axioms

$noOverlap(T1 : Type, \ T2 : Type, \ p1 : I8, \ p2 : I8) \ :=$
  $p1 \geq p2 + sizeof(T2) \ \parallel \ p2 \geq p1 + sizeof(T1)$
$embedded(T2 : Type, \ p1 : I8, \ p2 : I8) \ := \quad p1 \geq p2 \ \&\& \ p1 < p2 + sizeof(T2)$
$correctlyEmbedded(T1 : Type, \ T2 : Type, \ p1 : I8, \ p2 : I8) \ :=$
  match $T2$ **with**
  $\mid PrimitiveType \ => \ T1 == T2 \ \&\& \ p1 == p2$
  $\mid StructType \ =>$
    match $StructType$ **with**
    $\mid T \ t; \ StructType' \ => embeddedInField(T1, T, p1, p2) \parallel$
      $correctlyEmbedded(T1, StructType', p1, p2 + nextOffset(StructType))$
    $\mid T \ t \ => embeddedInField(T1, T, p1, p2)$
    end
  end
$embeddedInField(T1 : Type, \ T2 : Type, \ p1 : I8, \ p2 : I8) \ :=$
  match $T2$ **with**
  $\mid PrimitiveType \ => \ T1 == T2 \ \&\& \ p1 == p2$
  $\mid StructType \ =>$
    $nestedIn(T1, \ StructType) \ \&\& \ embedded(StructType, p1, p2)$
  end

Fig. 5. Helper functions for disjointness axioms

over the structure of arguments. Therefore, there will only be a finite number of instantiations, and the use of pattern based instantiation is complete for our axioms.

Together with the disjointness axiom, the memory representations of Section 5 are precise for all well-typed programs. Furhtermore, incorrect pointer arithmetic can be detected by automatically checking whether a pointer has a compatible type prior to every memory access.

7 Optimizations

7.1 Two-Phase Solving

After initial experiments, we observed that some of the constraint systems generated during our symbolic execution are particularly hard for the constraint

solver. Furthermore, we noticed that the solver mainly had a hard time when the constraints are unsatisfiable. When they are satisfiable, the constraint solver usually gives a solution fairly quickly.

This can be explained by analyzing the disjointness axioms. First of all, the axioms cause a quadratic number of disjointness constraints in the amount of pointers. Furthermore, each disjointness constraint is actually a disjunction of two inequalities. To make matters even worse, pointers are represented as bitvectors, therefore an inequality causes the creation of a circuit, i.e. a representation of the inequality by logical gates operating at the bit-level. Not surprisingly, the constraint systems give rise to a large number of case splits, especially when all, or at least many, cases have to be enumerated which often happens when the constraints are unsatisfiable.

To improve the performance, we split constraint solving in two phases:

- First, we perform a satisfiability check for a simplified set of constraints. In particular, all disjointness constraints are replaced by a single disequality between the two pointers (E.g. $noOverlap(T1, T2, p1, p2) \rightarrow p1 \mathrel{!=} p2$). The resulting constraint system is weaker than the original one. If the satisfiability check fails, there will not be a solution for the original constraint system and we can skip the second phase.
- Then, we exploit the incremental nature of the constraint solver and add the full disjointness axioms to the simplified constraint system. In principle, checking the full constraints first and only adding them when necessary is possible, but we did not implement such a scheme. In any case, the full constraints are necessary to remain precise.

7.2   Alignment

Data is said to be aligned when its address is divisible by certain powers of two. For example, on the X86 architecture, a 4-byte (8-byte) entity is aligned when its address is divisible by four (eight). Accessing misaligned data often imposes a performance penalty; it may even be forbidden. As a result, most compilers automatically align data structures according to their type by inserting padding bytes.

With alignment, two pointers with a primitive type will always be equal or not overlapping. For primitive types, we can exploit alignment by replacing the disjointness axiom for pointers of the same type by an alignment constraint (which states that the lower bits are equal to zero). The advantage is that the alignment constraint does not cause a quadratic number of pointer inequalities.

8   Evaluation

Since our approach aims at generating inputs for pointers as well, we evaluate it in the context of data structures where pointer reasoning is essential.

First, we test an implementation of red-black trees, a self-balancing binary search tree, taken from the Windows source code base. This is a challenging test

case because red black trees have complicated constraints over a data structure with pointers. For example, the sum of all black nodes on a path from the root to any leaf node is always the same. Furthermore, in this performance-optimized implementation the leafs of the red-black tree are represented by a sentinel node which is nested inside the tree itself. This was an excellent test to see if nested types are working correctly.

We have tested the red-black tree with both memory representations described in Section 5. To evaluate the overhead of the enforcement of the disjointness conditions, we manually created a version of the test case with the extra constraints that all pointers are aligned modulo 1024, which can be expressed efficiently by operations at the bit-level. As a consequence, different pointers never overlap. This trick was much more convenient to enforce the disjointness of the pointers than manually walking over the entire data structure and enforcing the disjointness constraints. Furthermore, these logical conditions are slightly more efficient than encoding the disjointnesses manually. Therefore, comparing the performance of the disjointness with this system is not completely fair, but it already gives a good idea whether our system is much slower because of the disjointness axioms or not. This technique has limited potential for automatisation because pointers to nested structs are not necessarily aligned.

When changing the constraint systems, the order in which execution paths are explored usually changes, since the next execution path depends on the test inputs computed by the constraint solver. In order to compare the performance of the different memory representations, we inserted a subtle bug in the fixup routine of the red-black trees. We stop the test input generation as soon as the first two test cases that trigger the bug have been reported.

The results can be seen in Figure 6. On the Y-axis, we report the number of tests that has been generated. The memory representation with a map per type clearly outperforms the representation with a map per primitive type. In Section 5, we mentioned that the first representation is potentially imprecise when it is impossible to statically know if an assignment to a pointer is an assignment to a field of a struct. In practice, this imprecision never occured while executing the red-black tree benchmark. A second observation is that the disjointness constraints do not have a big impact in the first memory representation (16s vs 28s). For the second representation though, it deteriorates the performance severely.

To test our optimizations, we have executed the red-black tree again using the first memory representation and all combinations of the optimizations. We again stopped the input generation when we found the first two failing test cases. The result can be seen in Figure 7. With two-phase solving, the test input generation is clearly faster. In combination with two-phase solving, alignment does not seem to offer too much improvement. Without two phase solving, alignment seems to deteriorate the performance, although this is hard to explain. In addition, we computed the average time for the constraint solver to handle a query. Without two-phase solving, the average is $.13s$. With two-phase solving, the averages for the first and the second phase are $.02s$ and $.05s$ respectively. These number
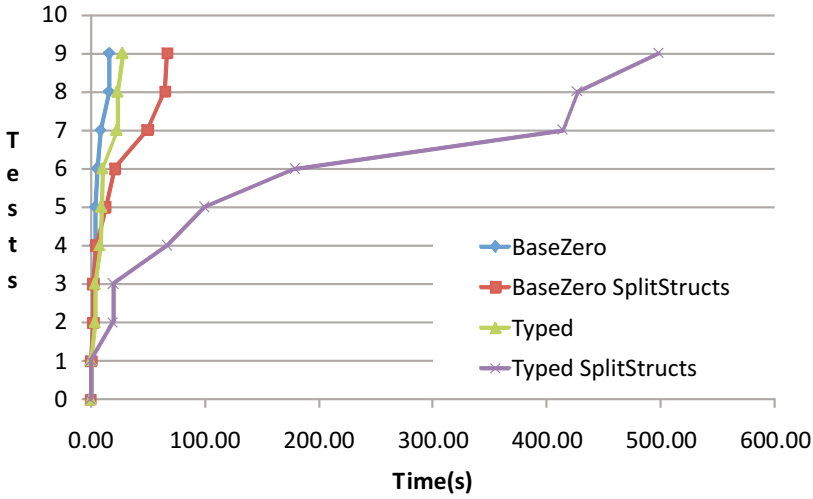
Fig. 6. Red black tree: Memory representations (*BaseZero* uses alignment modulo 1024 to enforce disjointness and *Typed* uses the the disjointness axioms. *SplitStructs* means that we only maintain a map per primitive type.)
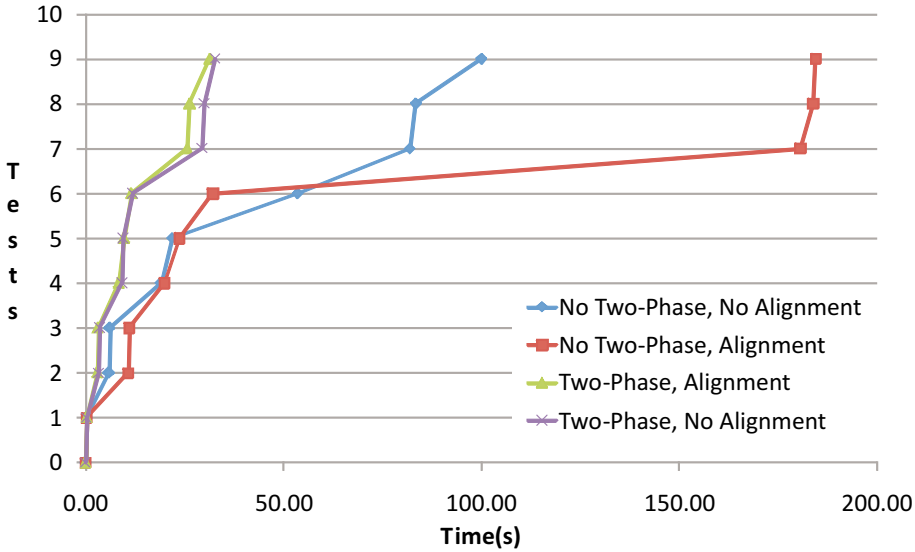

Fig. 7. Red black tree: Optimizations

confirm our hypothesis that we can improve the performance by doing a fast satisfiability check first.

Finally, we tested all combinations of the different options (memory representation, two-phase solving and alignment) on a linked list data structure. We used a fixed timeout of $600s$, and we report the size of the largest queue that has been

found over time. Figure 8 shows the results. Two-phase solving has the biggest impact on the size of the linked lists that are being generated. With two phase solving, the first memory representation seems to perform better. Without two phase solving, the second memory representation seems better. Also, alignment seems to improve the performance in three out of four cases.
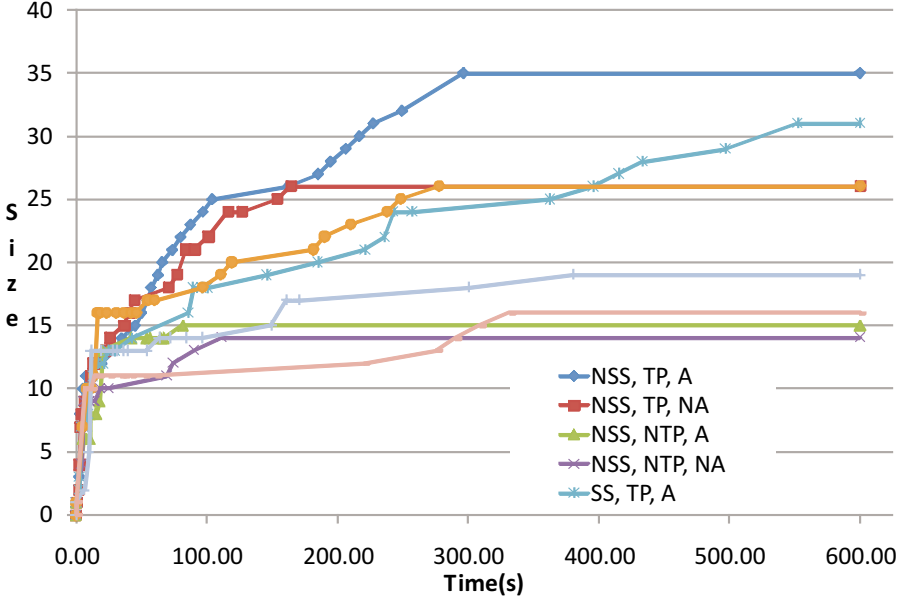
Fig. 8. Linked list: Optimizations (SS for SplitStructs, TP for two-phase solving, A for alignment and N for their negated counterparts)

9   Related Work

Recently, a broad range of test input generation tools have been developed based on symbolic execution [5,11,3,12,4,2]. We compare our work with them based on two dimensions:

Generating pointers as input. As we discussed in Section 3, most tools [5,11,3,12] focus on integration testing and do not support pointers as symbolic input for test methods (e.g. Sage [5] only focuses on files as input, and KLEE [12] on command-line arguments and files). For unit testing complex data structures this support is essential.

Some argue that a test that requires complex test inputs can be wrapped in another test that only takes (an array of) inputs with primitive types; the wrapping test first parses the complex data from the primitive data, and then calls the original test; this approach complicates the exploration of the code under test by requiring the exploration of the parser in addition, and it

also does not take into account possibly legal configurations of the complex data which are not generated by the parser.

An alternative solution is test sequence generation, the process of creating the data structure iteratively by starting with a constructor call followed with a list of methods with symbolic inputs. Unfortunately, test sequence generation is far from scalable in practice.

In this paper, we use a designated function to validate the data structure (also known as repOk methods). This function is similar to the concept of invariant in deductive software verification. For a valid test input, we assume that the data structure is valid, then we apply the function we want to test. Finally, we assert that the resulting data-structure is still valid.

Reasoning about pointers. There is much diversity in the way different tools handle pointers: e.g. Java Pathfinder [2] only supports object references.

Cute [4] does support pointers, they collect only (dis)equality constraints over these pointers. Although one might argue that using complicated operations on pointers is bad practice, it is used in rare but intricate cases. For example, we frequently encountered alignment-checks on pointers, where the lower bits of the pointer are inspected by the program. Our approach treats pointers as regular integers supporting all integer arithmetic operations.

SimC [13] is another tool to generate test inputs that support pointer reasoning. Unlike us, they model memory as one large array. SimC implicitly assumes that pointers do not overlap in incorrect ways.

Most tools perform concretization at some level to support pointer reasoning [3,5,4], i.e. they use the concrete value of a pointer as observed during concrete execution. Cute does not use the theory of arrays for representing memory but concretizes indices of array accesses. This leads to the inability to generate test inputs where $i == j$ in the following program: *a[i]=0; a[j]=1; if (a[i]==0) ERROR*. EXE concretizes pointers when there are double dereferences. Finally, Sage uses concretization to handle symbolic indexing into an array. We do not perform concretization to deal with pointers.

For program verification (of low level code), dealing with pointers is challenging as well. Most verification tools are incomplete with respect to pointers. For example, in VCC [14], pointers are treated as logical references instead of integers. Havoc [15] also treats pointers as integers. Havoc also has an encoding of the type system for SMT solvers [16], but our encoding is more precise. In particular, in Havoc, two different structs with fields of the same type can not be differentiated. Furthermore, they don't encode disjointness constraints at the byte level. Finally, separation logic [17] is a promising alternative way to reason about heap manipulating programs.

10   Conclusion

In this paper, we proposed a novel solution for generating test inputs for programs with pointers. We exploited the type information to encode disjointness

assumptions that characterize acceptable configurations of typed pointers in byte-addressable memory as constraints for the solver. As a result, the constraint solver only computes relevant heap shapes for the program under test.

We have implemented our approach in Pex, and evaluated it on red black trees and linked lists. From the two memory representations we created, the representation with a map per type was much faster than the representation where we only maintained a map per primitive type. Thanks to the two-phase solving optimization, the disjointness axioms have only minimal impact on the performance.

References

1. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
2. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In: ISSTA (2004)
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS 2006 (2006)
4. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. of ESEC/FSE 2005, pp. 263–272. ACM Press, New York (2005)
5. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of NDSS 2008 (Network and Distributed Systems Security) (2008)
6. Tillmann, N., de Halleux, J.: Pex–white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Notices 40(6), 213–223 (2005)
8. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Abstract synergy: A new algorithm for property checking (2006)
9. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: SOSP (2005)
10. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 2–23. Springer, Heidelberg (2005)
12. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008 (to appear)
13. Xu, Z., Zhang, J.: A test data generation tool for unit testing of c programs. QSIC 0, 107–116 (2006)
14. Schulte, W., Xia, S., Smans, J., Piessens, F.: A glimpse of a verifying c compiler – extended abstract (2007)
15. Chatterjee, S., Lahiri, S.K., Qadeer, S.: 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)
16. Condit, J., Hackett, B., Lahiri, S., Qadeer, S.: Unifying type checking and property checking for low-level codes. In: POPL (to appear, 2009)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)

Specification Mining with Few False Positives

Claire Le Goues and Westley Weimer⋆

University of Virginia
{legoues,weimer}@virginia.edu

Abstract. Formal specifications can help with program testing, optimization, refactoring, documentation, and, most importantly, debugging and repair. Unfortunately, formal specifications are difficult to write manually, while techniques that infer specifications automatically suffer from 90–99% false positive rates. Consequently, neither option is currently practical for most software development projects.

We present a novel technique that automatically infers partial correctness specifications with a very low false positive rate. We claim that existing specification miners yield false positives because they assign equal weight to all aspects of program behavior. By using additional information from the software engineering process, we are able to dramatically reduce this rate. For example, we grant less credence to duplicate code, infrequently-tested code, and code that exhibits high turnover in the version control system.

We evaluate our technique in two ways: as a preprocessing step for an existing specification miner and as part of novel specification inference algorithms. Our technique identifies which input is most indicative of program behavior, which allows off-the-shelf techniques to learn the same number of specifications using only 60% of their original input. Our inference approach has few false positives in practice, while still finding useful specifications on over 800,000 lines of code. When minimizing false alarms, we obtain a 5% false positive rate, an order-of-magnitude improvement over previous work. When used to find bugs, our mined specifications locate over 250 policy violations. To the best of our knowledge, this is the first specification miner with such a low false positive rate, and thus a low associated burden of manual inspection.

1   Introduction

Debugging, testing, maintaining, optimizing, refactoring, and documenting software are costly and time-consuming processes, yet they remain critically important: deployed programs with incorrect behavior cost billions of dollars and multiple lives each year [28]. Modifying existing code, correcting defects, and otherwise evolving software are major parts of maintenance [31], which is reported

to consume up to 90% of the total cost of software projects [32]. Incomplete documentation is a key maintenance difficulty [12]: up to 60% of maintenance time is spent studying existing software (e.g., [29, p.475], [30, p.35]). Understanding correct software behavior is central to maintaining, changing, and correcting code. Human processes and especially tool support for these activities depend on formal specifications of proper program behavior (e.g., [26]). Unfortunately, while low-level program annotations are becoming more and more prevalent [11], formal specifications remain rare.

Formal specifications are difficult for humans to construct [9], and incorrect specifications are difficult for humans to debug and modify [3]. Specification mining projects attempt to address these problems by inferring specifications from program source code or execution traces [1,2,15,17,33,37,38]. Unfortunately, existing techniques typically produce imprecise specifications and suffer from false positive rates of 90–99% [36] — that is, a very large proportion of candidate specifications produced by these techniques are not true program specifications. Some miners require that every inferred policy be corrected manually [3].

Specification mining can be compared to learning the rules of English grammar by reading essays written by high school students; we propose to focus on the essays of passing students and be skeptical of the essays of failing students. We claim that existing miners have high false positive rates in large part because they treat all code equally, even though not all code is created equal. For example, consider an execution trace through a recently modified, rarely-executed piece of code that was copied-and-pasted by an inexperienced developer. We argue that such a trace is a poor guide to correct behavior when compared with a well-tested, infrequently-changed, and commonly-executed trace.

The problem of mining temporal safety policies is undecidable in general [2], as it is impossible to learn regular languages in the limit [18, Theorem 1.8] based on finitely many examples. Existing miners thus use heuristics to decide which specifications are likely true. Our algorithm is no different in that regard; we infer temporal safety properties of the form "b must follow a," using heuristics based on information gleaned from the software engineering process.

We propose a new automatic specification miner that uses artifacts from software engineering processes to capture the trustworthiness of its input traces. The main contributions of this paper are:

- A set of source-level features related to software engineering processes that capture the trustworthiness of code for specification mining. We analyze the relative predictive power of each of these features.
- Empirical evidence that our notions of trustworthy code serve as a basis for evaluating the trustworthiness of traces. We provide a characterization for such traces and show that off-the-shelf specification miners can learn just as many specifications using only 60% of traces.
- A novel automatic mining technique that uses our trust-capturing features to learn temporal safety specifications with few false positives in practice. We evaluate it on over 800,000 lines of code and explicitly compare it to two previous approaches. Our basic mining technique learns specifications that

locate more safety-policy violations than previous miners (740 vs. 426) while presenting far fewer false positive specifications (107 vs. 567). When focused on precision, our technique obtains a low 5% false positive rate, an order-of-magnitude improvement on previous work, while still finding specifications that locate 265 violations. To our knowledge, this is the first specification miner that produces multiple candidate specifications and has a false positive rate under 90%.

The rest of this paper is organized as follows. In Section 2 we describe temporal safety specifications and highlight uses. Section 3 gives a brief overview of specification mining. Section 4 describes our approach to specification mining, including the code trustworthiness metrics used (Section 4.1). In Section 5 we present experiments supporting our claims and evaluating the effectiveness of our miner. We discuss related work in Section 6 and conclude in Section 7.

2   Temporal Safety Specifications

A *partial-correctness temporal safety property* is a formal specification of an aspect of correct program behavior [23], typically describing how to manipulate certain important resources and interfaces. Specifications take the form of finite-state machines that encode valid sequences of *events* relating to those resources that occur during the program's exeuction. For example, one event may represent reading untrusted data over the network, another may represent sanitizing it, and a third may represent a database query. Figure 2 shows such a specification for SQL injection attacks [25], based on the code in Figure 1.

Typically, each important resource is tracked separately [13]. Each finite state machine starts in its start state. A program *conforms* to a specification if and only if it terminates with the corresponding state machine in an accepting state. Otherwise, the program *violates* the specification. Such specifications can describe properties such as locking [10], resource leaks [36], security [25], high-level invariants [16] and memory safety [19], and more specialized properties such as the correct handling of `setuid` [9] or asynchronous I/O request packets [5]. These partial correctness specifications are distinct from and complementary to full formal behavior specifications.

```
void bad(Socket s, Conn c) {        void good(Socket s, Conn c) {
  string message = s.read();          string message = s.read();
  string query = "select * " +       c.prepare("select * from "
    "from emp where name = " +          + " emp where name = ?",
    message;                            message);
  c.submit(query);                    c.exec();
  s.write("result = " +               s.write("result = " +
    c.result());                        c.result());
}                                   }
```

Fig. 1. Pseudocode for an example internet service. The `bad` method passes untrusted data to the database; `good` works correctly. Important *events* are italicized.

These types of specifica-
tions can be used by almost
any existing defect-finding tool
(e.g., [5,10,11,16]); indeed, all
bug-finders require implicit or
explicit specifications. Formal
specifications can also help with

Fig. 2. Example specification for Figure 1

program testing [4], optimization [24], refactoring [21], documentation [6], and
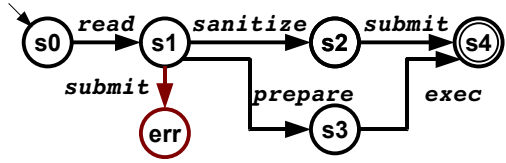repair [34]. Formal specifications are rare in practice, but not due to a lack of
possible uses.

3 Specification Mining

The goal of *specification mining* is to construct a formal specification using
examples of program behavior [2]. Traces of program behavior can be collected
from the source code (e.g., [15]) or from instrumented executions on indicative
workloads (e.g., [37]). These traces usually take the form of a sequence of function
calls. A specification miner examines such traces and produces one or more
candidate specifications, which must then be verified by a human.

Existing specification miners fall into two categories. Some produce a single
finite automaton policy with many states [1,2,37], while others produce many
small automata [15,17,36,38], typically of a fixed form. We focus on the latter,
because large automata are much more difficult to verify or debug [3].

The simplest and most common type of temporal specification is a two-state fi-
nite state machine [15,36]. Such two-state specifications require that event a must
always be followed by event b, correspond to the regular expression $(ab)^*$, and are
written $\langle a,b\rangle$. Examples include $\langle$open,close$\rangle$, $\langle$malloc,free$\rangle$, and $\langle$lock,unlock$\rangle$.
Such specifications often describe resource allocation or the correct restoration
of invariants, and are prevalent in practice. Even when attention is restricted to
two-state specifications, mining remains difficult [17].

How a specification miner should decide what event pairs constitute a valid
policy is non-obvious, especially in the face of red herrings such as $\langle$print,print$\rangle$,
or even policy violations. Engler *et al.* note that programmer errors can be
inferred by assuming the programmer is *usually* correct [15]. That is, common
behavior implies correct behavior. Engler *et al.*'s ECC miner counts the number
of times a and b appear together in order and the number of times that event a
appears without event b, and uses the z-statistic to rank the likelihood that the
correlation is deliberate on the part of the programmer. Their miner presents a
ranked list of candidate specifications to the programmer for inspection. Without
human guidance (e.g., without lists of important functions to focus on [15]), this
technique is prone to a very high rate of false positives. On one million lines
of Java code, only 13 of 2808 positively-ranked specifications generated by ECC
were real: a 99.5% false positive rate [36].

In previous work, we observed that programmers often make mistakes in
rarely-tested error handling code [36]. Tracking a single bit of information per
trace — whether that trace corresponded to a program error or not — improved

the mining accuracy dramatically, by an order of magnitude. We also included a software engineering consideration, restricting attention to specifications in which the events a and b came from the same package or library. We assumed that independent libraries, potentially written by separate developers, are unlikely to depend on each other for correctness at the API level. These insights reduced the number of candidates presented to the programmer by a large factor: on the same million lines of Java code, the WN miner generated only 649 candidate specifications, of which 69 were real, for an 89% false positive rate. However, this rate is still too high to be considered automatic; before being used, the candidate specifications must still be hand-validated.

4   Our Approach: Code Trustworthiness

We call code *trustworthy* if it is unlikely to exhibit API policy violations. Previous approaches have implicitly assumed that all execution traces are equally indicative of correct program behavior, that is, that all traces should be trusted equally. In this paper, we demonstrate that this assumption is incorrect. We present a specification miner that works in three stages:

1. Statically estimate the trustworthiness of each code fragment.
2. Lift that judgment to traces by considering the code visited along a trace.
3. Weight the contribution of each trace by its trustworthiness when counting event frequencies for specification mining.

We hypothesize that code is most trustworthy when it has been written by experienced programmers who are familiar with the project at hand, when it has been well-tested, and when it has been mindfully written (e.g., rather than copied-and-pasted). Previous work has found more errors in recently-changed code [27], unreadable code [7] and rarely-tested code [36]. Such information can be collected from a program's source code and version control history. Section 4.1 describes the set of features we have chosen to approximate the "trustworthiness" of code. Section 4.2 describes our mining algorithm in more detail.

4.1   Trustworthiness Metrics

Our goal is to automatically distinguish between code that is likely to adhere to program specifications and code that is not. Our specification miner thus uses a number of metrics to approximate the trustworthiness of code. We only consider metrics that can be computed automatically using commonly-available software engineering artifacts, such as the source code itself or version control information. In the interest of automation, we do not consider features that require manual annotation or human guidance. We use the following metrics:

Code Churn. Previous work has shown that frequently modified code is less likely to be correct [27]; changing the code to fix one defect often introduces another. We hypothesize that so-called churned code is less likely to adhere to specifications. Using version control information, we measure the time between

the current revision and the last revision for each line of code in wall clock hours. Similarly, we measure the total number of revisions to each line.

Author Rank. We hypothesize that some developers have a better understanding of the implicit specifications for a project than others. A senior developer who has performed many edits may remember more of the program invariants than a developer recently added to the group. Source control repositories track the author of each change. The *rank* of an author is the proportion of all changes committed to the repository that were committed by that author. We measure the author rank of the last author to touch each line of code.

Copy-Paste Development. We hypothesize that duplicated code is more error-prone because it has not been specialized to its new context and because patches to the original may not have propagated to the duplicate. We further hypothesize that duplicated code does not represent an independent correctness argument on the part of the developer; if `printf` follows `iter` in 10 duplicated code fragments, it is not 10 times as likely that ⟨iter,printf⟩ is a real specification. We measure repetition using the open-source `PMD` toolkit's copy-paste detector, which is based on the Karp-Rabin string matching algorithm [20].

Code Readability. In previous work, we showed that more readable code is less likely to contain errors [7]. We hypothesize that more readable code is thus also more likely to adhere to specifications. We measure code readability using our software readability metric, which is based on textual source code features and agrees with human annotators [7].

Path Feasibility. Infeasible paths are an artifact of the static enumeration process; we claim that they do not encode programmer intentions. Previous work has argued that it is always helpful to have more traces, even incorrect ones [35]; our experiments suggest that quality is more important than quantity (see Section 5.2). Merely excluding infeasible paths confers some benefit. However, we further hypothesize that infeasible paths suggest pairs that are *not* specifications. If the programmer has made it impossible for b to follow a along a path, ⟨a,b⟩ is unlikely to be required. We measure the feasibility of a path using symbolic execution; a path is infeasible if an external theorem prover (in our case, Simplify) reports that its symbolic branch guards are inconsistent.

Path Frequency. We theorize that paths that are frequently executed by indicative workloads and testcases are more likely to contain correct behavior. We use a research tool that can statically estimate the relative runtime frequency of a given path through a program [8] to measure path frequency. We measure relative runtime frequency with respect to the enclosing method.

Path Density. We hypothesize that a method with few static paths is likely to exhibit correct behavior and that a method with many paths is likely to exhibit incorrect behavior along at least one of them. We define "path density" as the number of traces it is possible to enumerate in each method and in each class. A low path density for traces containing the paired events ab and a high path density for traces that contain only a both make ⟨a,b⟩ a likely specification.

4.2   Mining Algorithm Details

Our mining algorithm extends our previous WN miner [36]. Formally, our miner takes as input:

1. The program source code P. The variable ℓ ranges over source code locations.
2. A set of trustworthiness metrics $M_1 \dots M_q$, with $M_i(\ell) \in \mathbb{R}$.
3. A set of important events Σ, typically taken to be all of the function calls in P. We use the variables a, b, etc., to range over Σ.

Our miner produces as output a set of candidate specifications $C = \{ \langle a,b \rangle \mid a$ should be followed by $b\}$. We determine the validity of a particular candidate specification by manual inspection; we present experimental results in Section 5.

Our algorithm first statically enumerates traces through P. Since there are an infinite number of traces, we must choose a finite enumeration strategy. We consider each method m in P in turn. Using a breadth-first traversal, we enumerate the first k paths through m, assuming that branches can either be taken or not and that an invoked method can either terminate normally or raise any of its declared exceptions [36]. We pass through loops no more than once. This produces a set of traces T, where each trace t is a sequence of program locations ℓ. We write $a \in t$ if the event a occurs in trace t and $a \dots b \in t$ if the event a occurs and is followed by the event b in that trace. We also note whether or not a trace involves exceptional control flow; we write $Error(t)$ for this judgment [36].

Next, where necessary, our miner lifts trustworthiness metrics from locations to traces. Our lifting is parametric with respect to an aggregation function $A : \mathcal{P}(\mathbb{R}) \to \mathbb{R}$. We use the functions max, min, span and average in practice. We write M^A for a trustworthiness metric M lifted to work on traces: $M^A(t) = A(\{M(\ell) \mid \ell \in t\})$. We write $\mathcal{M}$ for the metric lifted again to work on sets of traces: $\mathcal{M}(T) = A(\{M^A(t) \mid t \in T\})$.

Finally, we consider all possible candidate specifications. For each a and b in Σ, we collect a number of *features*. We write N_{ab} for the number of times a is followed by b in a normal (non-error) trace. We write N_a for the number of times a occurs in a normal trace, with or without b. We similarly write E_{ab} and E_a for counts in error traces. We write $SP_{ab} = 1$ when a and b are in the same package (i.e., defined in the same library). We write $DF_{ab} = 1$ when a and b are connected by dataflow information: when every value and receiver object expression in b also occurs in a [36, Section 3.1].

In previous work we showed that both the ECC and WN miners can be expressed using this set of features [35]. The ECC miner returns $\langle a,b \rangle$ when a is followed by b in some traces but not in others: $N_a - N_{ab} + E_a - E_{ab} > 0$ and $N_{ab} + E_{ab} > 0$ and $DF_{ab} = 1$. The WN miner returns $\langle a,b \rangle$ when $E_{ab} > 0$ and $E_a - E_{ab} > 0$ and $DF_{ab} = SP_{ab} = 1$. Both of these miners encode arbitrary heuristic choices about which features are considered, the relative importance of various features, and which features must have high values.

We extend the set of features by adding the aggregate trustworthiness for each lifted metric M^A. We write $\mathcal{M}_{iab}$ (resp. $\mathcal{M}_{ia}$) for the aggregate metric values on the set of traces that contain a followed by b (resp. contain a). Figure 3 lists the

$$
\begin{aligned}
N_a &= |\{t \mid a \in t \ \wedge \ \neg Error(t)\}| \\
N_{ab} &= |\{t \mid a \ldots b \in t \ \wedge \ \neg Error(t)\}| \\
E_a &= |\{t \mid a \in t \ \wedge \ Error(t)\}| \\
E_{ab} &= |\{t \mid a \ldots b \in t \ \wedge \ Error(t)\}| \\
SP_{ab} &= 1 \text{ if } a \text{ and } b \text{ are in the same package, } 0 \text{ otherwise} \\
DF_{ab} &= 1 \text{ if every value in } b \text{ also occurs in } a, 0 \text{ otherwise} \\
\mathcal{M}_{ia} &= \mathcal{M}_i(\{t \mid a \in t\}) \qquad \text{where } \mathcal{M}_i \text{ is a lifted trustworthiness metric} \\
\mathcal{M}_{iab} &= \mathcal{M}_i(\{t \mid a \ldots b \in t\}) \ \text{ where } \mathcal{M}_i \text{ is a lifted trustworthiness metric}
\end{aligned}
$$

Fig. 3. Features used by our miner to evaluate a candidate specification $\langle a,b \rangle$

set of features considered by our miner when evaluating a candidate specification $\langle a,b \rangle$. Since we have multiple aggregation functions and metrics (see Section 4.1), $\mathcal{M}_{ia}$ actually corresponds to over a dozen individual features.

We also include a number of statistical features, fractions and percentages related to the main frequency counts $N_a \ldots E_{ab}$, such as the z-statistic used by ECC to rank candidate specifications; we thus use over 30 total features f_i for each pair $\langle a,b \rangle$. Rather than asserting an *a priori* relationship between these features that candidate specifications must adhere to, we use linear regression to learn a set of coefficients c_i and a cutoff *cutoff*, such that our miner outputs $\langle a,b \rangle$ as a candidate specification iff $\sum_i c_i f_i < cutoff$. This involves a training stage to determine both the coefficients and the cutoff, described in detail in Section 5.

5   Experiments

We evaluate our miner on several open-source Java benchmarks, shown in Figure 4. We selected these programs to allow a direct comparison to previous work [17,35,36,38]. We restricted attention to programs with CVS or SVN source-control repositories. For each program, we statically enumerated traces (up to a limit of 20 per method) and gathered the required information for the

Program Version	LOC	Description
`hibernate2 2.0b4`	57k	Object persistence
`axion 1.0m2`	65k	Database
`hsqldb 1.7.1`	71k	Database
`cayenne 1.0b4`	86k	Object persistence
`jboss 3.0.6`	107k	Middleware
`mckoi-sql 1.0.2`	118k	Database
`ptolemy2 3.0.2`	362k	Design modeling
Total	866k	

Fig. 4. Benchmarks used in our experiments

trustworthiness metrics described in Section 4.1. We do not need source code *implementing* a particular interface; instead, we generate traces from the client code that *uses* that interface (as in [2,14,17,38]). One expensive operation was computing path feasibility, which required multiple calls to Simplify, an external theorem prover. On a 3 GHz Intel Xeon machine, computing it on the `mckoi-sql` (our second-largest) benchmark took 25 seconds. Enumerating all static traces for `mckoi-sql`, with a maximum of 20 traces per method, took 912 seconds in total; this happens once per program. Collecting the other metrics for `hsqldb`

is relatively inexpensive (e.g., 6 seconds for readability, 7 seconds for path frequency). The actual mining process (i.e., considering the features for every pair of events in `mckoi-sql` against the cutoff) took 555 seconds. The total time for our technique was about 30 minutes per 100,000 lines of code.

5.1   Trustworthiness Metrics: Learning Cutoffs and Coefficients

First, we learn the coefficients and cutoff that determine which candidate specifications to output, and thus the relative importance of our trustworthiness metrics. We use *recall* and *precision* to evaluate potential coefficients. Recall is the number of real specifications returned out of all possible real specifications, or the probability that a real specification is returned by the algorithm. Precision is the fraction of candidate specifications that are not false positives. A high recall indicates that the miner is doing useful work (i.e., returning real specifications), but without a corresponding high precision, those real specifications will be drowned in a sea of false positives. We claim that current false positive rates are too high for existing techniques to be of practical use.

We use linear regression to find the coefficients for our miner. Linear regression requires annotated answers (i.e., a set of known-valid and known-invalid specifications). We use the valid and invalid specifications mined and described in previous work [35,36] as a training set. Given the set of linear regression coefficients, we perform a linear search of possible cutoffs and choose the one that maximizes an objective function. That objective function can be the harmonic mean of precision and recall (our *normal miner*) or just precision (which yields a *precise miner* with very few false positives).

Metric	F	p
Frequency	32.3	0.0000
Copy-Paste	12.4	0.0004
Code Churn	10.2	0.0014
Density	10.4	0.0013
Readability	9.4	0.0021
Feasibility	4.1	0.0423
Author Rank	1.0	0.3284

Fig. 5. Analysis of variance

Our first experiment **evaluates the relative importance of our trustworthiness metrics**. A per-feature analysis of variance, over all of the training data, is shown in Figure 5. The F column denotes the F-ratio, or the square of the variance explained by the feature over the variance not explained. It is near 1 if the feature does not affect the model. The p column shows the probability that the feature does not affect the miner.

All features except Author Rank had a significant main effect ($p \leq 0.05$). The Frequency metric, encoding our static prediction of how often the path would be executed at run-time [8], was our most important feature: commonly-run (and thus well-tested) paths do not demonstrate erroneous behavior. All of our new trustworthiness features were more important to mining than feasibility, which is, to our knowledge, the only one that had been previously investigated [1]. We were surprised to discover that our formulation of author rank had no effect on the model: whether the last person to touch a line of code was a frequent contributor to the project is not related to whether traces adhered to specifications.

5.2   Trust Matters for Trace Quality

In our second experiment, we **demonstrate that our trustworthiness metrics improve existing techniques for automatic specification mining**. For each of our benchmarks, we run the unmodified WN miner [36] on multiple input trace sets. For generality, we restrict attention to feasible traces, since miners such as JIST already disregard infeasible paths [1].

We compare WN's performance on a baseline set of feasible static traces to its performance on trustworthy subsets of those traces. For this experiment we define the trustworthiness of a trace to be a linear combination of the metrics from Section 4.1, with coefficients based on their relative predictive power for specification mining (the F column in Figure 5).

On the entire baseline set, WN miner produces 75 real specifications. Averaged over all the benchmarks, WN finds the same specifications using only the top 60% most trustworthy traces: 40% of the traces can be dispensed with while preserving true positive counts. As a point of comparison, when a random 40% of the traces are discarded, we find only 56 true specifications in total, with a 4% higher rate of false positives.

We also explore the impact of trustworthy traces on false positive rates by passing various proportions of trustworthy input to the WN miner. Figure 6 shows the results when only the 25% most trustworthy traces are used. On the baseline set, WN has 683 false positives: a false positive rate of 90%. When restricted to the 25% most trustworthy traces, WN produces 39 real specifications and 306 false positives: a false positive rate of 89%. Notably, we find over one-half of the specifications with only one-fourth of the input, without sacrificing the false positive rate. Beyond halving the raw false positive rate, and thus human effort required to validate the results, this is useful if the smaller output set contains
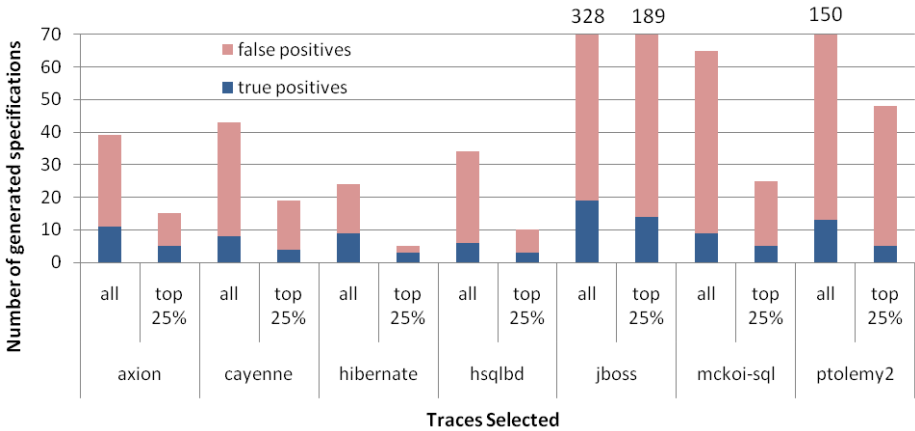

Fig. 6. The false positive rate of the off-the-shelf WN miner on various input sets. The total height of the bar represents the number of candidate specifications returned to the user for inspection.

particularly helpful specifications, which we investigate next. As a lower bound, only two true specifications can be mined from the 25% *least* trustworthy traces.

Any static specification mining technique involves a particular trace enumeration strategy; trace generation is often a bottleneck. Rather than enumerating a certain number of traces per method, we claim that trustworthy traces should be pursued and untrustworthy traces should be skipped. These results also have implications for multi-party techniques to mine specifications collaboratively by sharing trace information [35]. Focus should be placed on sharing information from trustworthy traces. Our trustworthiness metrics could generally be used as a preprocessing step to improve any static trace-based specification miner (e.g., [15,17,36,38]). However, they can be even more useful when directly incorporated into a mining algorithm.

5.3   Trustworthy Specification Mining

For our main experiment, we **measure the efficacy of our new specification miner** on all input trace sets. We must first verify that our miner is not biased with respect to our training data. A potential threat to the validity of our results is over-fitting by testing and training on the same data. We use 10-fold cross validation to mitigate this threat [22]. We randomly partition the data into 10 sets of equal size. We test on each set in turn, training on the other nine; in this way we never test and train on the same data. If the average results of cross-validation (over many random partitionings) are different from the original results, it may indicate bias. For our experiment, the difference was less than 0.01%, indicating little or no bias.

Figure 7 shows the results of applying our new specification miner from Section 4 to the benchmarks in Figure 4. For each benchmark, we report the number of candidate specifications returned, broken down into valid specifications and false positives (determined by manual verification of the results). We

Program	Normal Miner			Precise Miner			WN			ECC		
	Specs	False	Bugs	Specs	False	Bugs	Specs	False	Bugs	Specs	False	Bugs
`hibernate`	7	8 = 53%	279	5	1 = 17%	153	9	42 = 82%	93	3	421 = 99%	21
`axion`	7	5 = 42%	71	4	0 = 0%	52	8	17 = 68%	45	0	96 = 100%	0
`hsqldb`	3	1 = 25%	36	1	0 = 0%	5	7	55 = 89%	35	0	244 = 100%	0
`jboss`	14	75 = 84%	255	2	0 = 0%	12	11	103 = 90%	94	2	442 = 99%	4
`cayenne`	5	7 = 58%	45	3	0 = 0%	23	5	30 = 86%	18	3	308 = 99%	8
`mckoi-sql`	7	10 = 59%	20	2	0 = 0%	7	19	137 = 88%	69	2	344 = 99%	5
`ptolemy`	6	1 = 14%	44	3	0 = 0%	13	9	183 = 95%	72	3	653 = 99%	12
Total	49	107 = 69%	740	20	1 = 5%	265	68	567 = 89%	426	13	2508 = 99%	50

Fig. 7. Comparative mining results on 800kLOC. "Specs" indicates valid specifications, "False" indicates false positive specifications. "Bugs" totals, for each valid specification found, the number of distinct methods that violate it. The two left headings give results for our Normal Miner and our Precise Miner; WN and ECC are previous algorithms.

also report the number of distinct methods that violated the valid mined specifications (i.e., the number of policy violations found by using that specification with a bug-finding tool). Each method is counted only once per specification, even if multiple paths through that method violate it. We reprint published results for the WN [36] and ECC [15] miners for comparison. Recall that our normal miner minimizes both false positives and false negatives, while our precise miner only minimizes false positives (see Section 5.1).

The WN and ECC miners were chosen for comparison because of their comparatively low false positive rates. Other methods produce even more candidates. On jboss, the Perracotta miner produces 490 candidate two-state properties, which the authors say "is too many to reasonably inspect by hand." [38] Gabel and Su report mining over 13,000 candidates from hibernate [17]. Our precise miner produces six – one is a false positive, and the other five find over 150 violations.

Our normal miner finds important specifications with a low false positive rate. It improves on the false positive rate of WN by 20%. Moreover, the specifications that it finds generally find more violations than those found by WN: 740 violations, or 15 per valid specification, compared to WN's 426, or 7 per valid specification. However, the end-user inspects both valid and invalid specifications. Each candidate specification from our miner helps to find 4 violations on average; for WN, less than 1 violation is found on average per candidate inspected.

This precise miner finds fewer valid specifications, but its 5% false positive rate approaches levels required for automatic use. It finds 30% as many specifications as WN, but 60% of the violations: each candidate inspected yields over 12 violations on average. Users are often unwilling to wade through voluminous tool output [15,19]; with a 5% false positive rate, and more useful specifications than those of previous work, we claim that our precise miner might be reasonable in both interactive and automatic settings.

5.4   Threats to Validity

Although our two miners outperform existing approaches in terms of bugs found and false positives avoided, our results may not generalize to industrial practice. The benchmarks used in this project may not be representative of other projects. We chose the benchmarks to be directly comparable with previous work [17,35,36,38], and note that the domains represented are more indicative of server and back-end computing than of client code. A second threat is overfitting. We use cross-validation in Section 5.3 to demonstrate that our results are not biased by over-fitting. A third threat lies in our manual validation of the output: our human annotation process may mislabel candidate specifications. To mitigate this threat we re-checked a fraction of our judgments at random and used the source code of a and b to evaluate $\langle a,b \rangle$. A final threat lies in our use of "bugs found" as a proxy for specification utility: while our mined specifications find more policy violations, they may not be as useful for tasks such as documenting or refactoring. We leave an investigation of specification utility for future work.

6   Related Work

Our work is most closely related to existing specification mining algorithms (see [36] for a survey). The `ECC` [15] and `WN` [36] algorithms are formalized in detail in Section 4.2. The `WML_static` [37] miner examines library source code, assumes that typestate is explicitly captured by object fields and thrown exceptions, and produces a single multi-state specification. The `WML_dynamic` [37] miner examines dynamic traces and produces a permissive multi-state specification that describes all observed behavior. The `JIST` [1] miner refines the `WML_static` approach and uses techniques from software model checking to rule out infeasible paths. The `Perracotta` [38] miner mines multiple candidate specifications that match a given template (e.g., the two-state specification form used in this paper is one such template). Gabel and Su [17] extend `Perracotta` using BDDs, show that two-state mining is NP-complete, and show that some specifications cannot be mined by composing multiple two-state specifications. The `Strauss` tool [2] uses probabilistic finite state machine learning to learn a single specification from traces. Shoham *et al.* [33] mine by using abstract interpretation where the abstract values are specifications.

Unlike `WML_static`, `JIST`, `Strauss` and Shoham *et al.*, we do not require that important parts of the specification, such as the classes of interest, be given in advance by the user. Unlike `Strauss`, `WML_dynamic`, `JIST`, and Shoham *et al.*, we produce multiple candidate specifications rather than a single specification; complex specifications are difficult to debug and verify [3]. Unlike `Perracotta` or Gabel and Su, we cannot mine more complicated templates, such as three-state specifications. Like `ECC`, `WN`, and Gabel and Su, our miner is scalable. The primary difference between our miner and previous miners is that we use software engineering information, encoded as trustworthiness metrics, to weight input traces and thus obtain low false positive rates. To our knowledge, no published miner that produces multiple candidates has a false positive rate under 90%; we present one with a 5% false positive rate that still finds over 250 violations.

7   Conclusion

Formal specifications have myriad uses, from testing and optimizing, to refactoring and documenting, to debugging and repair. Formal specifications are difficult to produce manually, and existing specification miners typically have 90–99% false positive rates. We claim that not all parts of a program are equally indicative of correct behavior. We encode this intuition using trustworthiness metrics such as predicted execution frequency, measurements of copy-paste code, code churn, software readability or path feasibility. These metrics can be used to improve the performance of existing trace-based miners by focusing on trustworthy traces: equivalent results can be obtained using only 60% of the input. We also use our metrics to create a new specification miner and compare it to two previous approaches on over 800,000 lines of code. Our basic miner learns specifications that locate hundreds more bugs than previous miners while presenting hundreds fewer false positive candidates. When focused on precision, our

technique obtains a low 5% false positive rate, an order-of-magnitude improvement on previous work, while still finding specifications that locate hundreds of violations. To our knowledge, among specification miners that produce multiple candidate specifications, this is the first to maintain a false positive rate under 90%. We believe it to be a first step towards utility in an automated setting.

References

1. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: POPL (2005)
2. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
3. Ammons, G., Mandelin, D., Bodík, R., Larus, J.R.: Debugging temporal specifications with concept analysis. In: Programming Language Design and Implementation, pp. 182–195 (2003)
4. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO, pp. 1–22 (2004)
5. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: European Systems Conference, April 2006, pp. 103–122 (2006)
6. Buse, R.P.L., Weimer, W.: Automatic documentation inference for exceptions. In: ISSTA, pp. 273–282 (2008)
7. Buse, R.P.L., Weimer, W.: A metric for software readability. In: ISSTA, pp. 121–130 (2008)
8. Buse, R.P.L., Weimer, W.: The road not taken: Estimating path execution frequency statically (2009)
9. Chen, H., Wagner, D., Dean, D.: Setuid demystified. In: USENIX Security Symposium, pp. 171–190 (2002)
10. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: ICSE, pp. 762–765 (2000)
11. Das, M.: Formal specifications on industrial-strength code—from myth to reality. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, p. 1. Springer, Heidelberg (2006)
12. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A study of the documentation essential to software maintenance. In: SIGDOC, pp. 68–75 (2005)
13. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: PLDI, pp. 59–69 (2001)
14. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Symposium on Operating Systems Design and Implementation (2000)
15. Engler, D.R., Chen, D.Y., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: SOSP, pp. 57–72 (2001)
16. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245 (2002)
17. Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: ICSE, pp. 51–60 (2008)
18. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)

19. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: OOPSLA Companion, pp. 132–136 (2004)
20. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31(2), 249–260 (1987)
21. Kataoka, Y., Ernst, M., Griswold, W., Notkin, D.: Automated support for program refactoring using invariants. In: ICSM, pp. 736–743 (2001)
22. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. IJCAI 14(2), 1137–1145 (1995)
23. Kupferman, O., Lampert, R.: On the construction of fine automata for safety properties. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 110–124. Springer, Heidelberg (2006)
24. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. SIGPLAN Not. 40(1), 364–377 (2005)
25. Livshits, V.B., Lam, M.S.: Finding security errors in Java programs with static analysis. In: USENIX Security Symposium, August 2005, pp. 271–286 (2005)
26. Malayeri, D., Aldrich, J.: Practical exception specifications. In: Advanced Topics in Exception Handling Techniques, pp. 200–220 (2006)
27. Nagappan, N., Ball, T.: Using software dependencies and churn metrics to predict field failures: An empirical case study. In: ESEM, pp. 364–373 (2007)
28. National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3 (May 2002)
29. Pfleeger, S.L.: Software Engineering: Theory and Practice. Prentice Hall PTR, Upper Saddle River (2001)
30. Pigoski, T.M.: Practical Software Maintenance: Best Practices for Managing Your Software Investment. John Wiley & Sons, Inc., Chichester (1996)
31. Ramamoothy, C.V., Tsai, W.-T.: Advances in software engineering. IEEE Computer 29(10), 47–58 (1996)
32. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices (2003)
33. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: ISSTA, pp. 174–184 (2007)
34. Weimer, W.: Patches as better bug reports. In: GPCE, pp. 181–190 (2006)
35. Weimer, W., Mishra, N.: Privately finding specifications. IEEE Trans. Software Eng. 34(1), 21–32 (2008)
36. Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
37. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: ISSTA (2002)
38. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: ICSE, pp. 282–291 (2006)

Path Feasibility Analysis for String-Manipulating Programs

Nikolaj Bjørner[1], Nikolai Tillmann[1], and Andrei Voronkov[2]

[1] Microsoft Research
http://research.microsoft.com
[2] University of Manchester
http://www.voronkov.com

Abstract. We discuss the problem of path feasibility for programs manipulating strings using a collection of standard string library functions. We prove results on the complexity of this problem, including its undecidability in the general case and decidability of some special cases. In the context of test-case generation, we are interested in an efficient finite model finding method for string constraints. To this end we develop a two-tier finite model finding procedure. First, an integer abstraction of string constraints are passed to an SMT (Satisfiability Modulo Theories) solver. The abstraction is either unsatisfiable, or the solver produces a model that fixes lengths of enough strings to reduce the entire problem to be finite domain. The resulting fixed-length string constraints are then solved in a second phase. We implemented the procedure in a symbolic execution framework, report on the encouraging results and discuss directions for improving the method further.

1 Introduction

Dynamic symbolic execution [8,3,14,16] has recently gained attention in the context of test-case generation. It extends static symbolic execution [11] by collecting symbolic constraints from concrete execution traces obtained by monitoring the executed instructions. In order to explore a different execution path it suffices to modify one of the extracted symbolic traces by selecting and negating a branch condition, which we call *flipping a branch*. Then a constraint solver is used to provide a satisfying assignment to the modified path condition.

Strings form a fundamental data type found in most if not all general purpose programming languages. Strings may be represented in various ways, such as a pointer to a 0-terminated array of characters (in C), as an array object with an explicit length (in Java and C#), or even as a singly linked list (in Haskell). Programs that manipulate strings can often abstract from the representation and use a set of library routines to perform common functions on strings, such as converting characters to strings, finding characters and extracting substrings.

The problem. String library routines are themselves implemented as programs, so dynamic symbolic execution can apply to string routines by exploring the underlying programs and solving constraints on the data types used in these programs. There is

an inherent overhead of this approach, as the general dynamic symbolic exploration engine has to search the state space of the string library routines for solutions to string constraints. We can take advantage of the fact that path constraints from common string library routines have a mathematical abstraction rooted in word equations. This suggests that we may work at the level of abstract strings and treat calls to the string library functions as operations in a *theory* of strings. As we will see, the full set of constraints that can be created from common string functions do not fall in a decidable class. On the other hand, our objective is really to find small strings that can be supplied as unit tests, so an incremental small and finite model-finding routine provides the right match.

The main existing way of handling strings in symbolic test-case generation tools is to *not* handle them specially. For example in the current release of Pex [16], strings are represented as arrays and string library routines are explored like any other procedure. As a result, simple calls to library functions become programs containing loops. Summaries [7] provide one additional layer on top of the string procedures to allow the search on feasible paths to coalesce several traversals of the same procedure body.

Example 1. Consider the program shown in Figure 1. This program checks whether the input string s is a URL encoding a query about EasyChair to either Microsoft Live Search or Google. Essentially, such a string must start with "http://" followed by a domain of one of the search engines, followed by a "/", and after this "/" it should contain a substring "EasyChair" and not contain other "/".

```
private bool IsEasyChairQuery(string str)
{
  // (1) check that str contains "/" followed by anything not
  // containing "/" and containing "EasyChair"
  int lastSlash = str.LastIndexOf('/');
  if (lastSlash < 0){return false;}
  var rest = str.Substring(lastSlash + 1);
  if (! rest.Contains("EasyChair")){return false;}
  // (2) Check that str starts with "http://"
  if (! str.StartsWith("http://")){return false;}
  // (3) Take the string between "http://" and the last "/".
  // if it starts with "www." strip the "www." off
  var t = str.Substring("http://".Length,
                        lastSlash - "http://".Length);
  if (t.StartsWith("www.")){t = t.Substring("www.".Length);}
  // (4) Check that after stripping we have either "live.com"
  // or "google.com"
  if (t != "live.com" && t != "google.com"){return false; }
  // s survived all checks
  return true;
}
```

Fig. 1. The EasyChair Query program

We will use this program as the running example. Consider the following path in the program, where queries are denoted by "?".

```
lastSlash = str.LastIndexOf('/');
? ¬ lastSlash < 0
rest = str.Substring(lastSlash + 1);
? rest.Contains("EasyChair")
? str.StartsWith("http://")
t = str.Substring("http://".Length,lastSlash-"http://".Length);
? t.StartsWith("www.")
t = t.Substring("www.".Length);
? t = "live.com"
```

We are interested in checking feasibility of this path, that is, finding an input string `str` so that the program run with this string as an input will follow this path.

The rest of this paper is organized as follows. Section 2 defines path feasibility in the context of constraints from basic .NET string library functions and defines a first-order string library language corresponding to these library functions. The resulting constraints for all but one library function are compiled into the core language as outlined in Section 3. Section 4 discusses the decidability of the library language. One fragment is equivalent to a long standing open problem in word equations, another fragment is shown undecidable. This confirms the complexity of the path feasibility problem for string library functions. We also show that a so-called *fixed-length* fragment is decidable: this fact is used in our implementation. We point out that two other decidable fragments can be obtained using results from word equations and automatic structures. Section 5 outlines our incremental procedure for enumerating solutions to core language constraints. Sections 6 and 7 describe an implementation and evaluation.

2   Path Feasibility and String Constraints

In this section we define a language for representing the path feasibility problem as a constraint satisfaction problem.

The String Library Language. In this section we will introduce a *string library language* $\mathcal{LL}$. This language is a first-order language for describing constraints involving string library functions. The string library we are interested in is the .NET String library, however, we believe that other string libraries can be captured by similar languages and processed using the methods described in this paper.

Let $\mathbb{C}$ be a finite set of *characters* and $\mathbb{S}$ the set of all strings built using these characters. By $\mathbb{I}$ we denote the set of all integers. The *string library language* $\mathcal{LL}$ is a many-sorted first-order language defined as follows. The language has the following sorts: the *character sort* $\mathcal{C}$, the *string sort* $\mathcal{S}$, the *integer sort* $\mathcal{I}$ and the Boolean sort $\mathcal{B}$. It also contains a countably infinite number of variables of each sort.

The language $\mathcal{LL}$ contains constants denoting all integers, characters and strings, and some functions on strings and integers. Table 1 contains a description of a representative subset of the .NET string library. Positions in strings are number from 0 so the character at the position 0 in a string is always the first character of the string.

Some functions of the .NET library are overloaded. To resolve ambiguity, we add indexes to the names of these functions. For example, there are six different .NET

Table 1. String library functions

function	type	meaning
$Chars(s, i)$	$\mathcal{S} \times \mathcal{I} \to \mathcal{C}$	the character at position i in s
$Compare(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \to \mathcal{I}$	string comparison
$Concat(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \to \mathcal{S}$	string concatenation
$Contains(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}$	true if s_2 is a substring of s_1
$Equals(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}$	true if $s_1 = s_2$
$IndexOf_4(s_1, s_2, i)$	$\mathcal{S} \times \mathcal{S} \times \mathcal{I} \to \mathcal{I}$	the index of the first occurrence of s_2 in s_1 starting at position i or -1 if not found
$LastIndexOf_1(s, c)$	$\mathcal{S} \times \mathcal{C} \to \mathcal{I}$	the index of the last occurrence of c in s or -1 if not found
$Length(s)$	$\mathcal{S} \to \mathcal{I}$	the length of s
$Replace_2(s_1, s_2, s_3)$	$\mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$	replace all occurrences of s_2 in s_1 with s_3
$StartsWith(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}$	true if s_1 starts with s_2
$Substring_1(s, i)$	$\mathcal{S} \times \mathcal{I} \to \mathcal{S}$	substring of s starting at position i
$Substring_2(s, i_1, i_2)$	$\mathcal{S} \times \mathcal{I} \times \mathcal{I} \to \mathcal{S}$	substring of s starting at position i_1 and having length i_2
$ToString(c)$	$\mathcal{C} \to \mathcal{S}$	string consisting of a single character c
$ToUpper(s)$	$\mathcal{S} \to \mathcal{S}$	string obtained by converting s to upper case

functions having the name *IndexOf*. In $\mathcal{LL}$ we add indexes to function names to distinguish them (cf. *IndexOf*$_4$ in the table).

To define the semantics of $\mathcal{LL}$ we introduce a notion of value assignment. A *value assignment* for this language is a mapping of variables to $\mathbb{C} \cup \mathbb{S} \cup \mathbb{I}$ so that every variable is mapped to an element of the corresponding domain, for example, variables of the sort $\mathcal{I}$ are mapped to integer values. We will now extend value assignments to arbitrary expressions and quantifier-free formulas. To this end we should take care of undefined values since some of the library functions are partial and calling them outside of their domain causes an exception. We will specify the exception conditions later.

Let us introduce a special *undefined value* $\bot$ and extend value assignments to arbitrary expressions of $\mathcal{LL}$ as follows. For every value assignment v and expression e different from a variable we define $v(e)$ as follows.

1. If e is a constant, then $v(e)$ is the value of this constant, for example the value of the integer constant 7 is the number 7.
2. Suppose that e has the form $f(e_1, \ldots, e_n)$ where f is a function of $\mathcal{LL}$ and let $v_1 = v(e_1), \ldots, v_n = v(e_n)$. If for some $i \in \{1, \ldots, n\}$ we have $v_i = \bot$, then $v(e) = \bot$. Otherwise, let the $\widehat{f}$ be the function corresponding to f (see Table 1 for the definition of these functions). If $\widehat{f}$ is undefined on $(v_1, \ldots, v_n)$, then $v(e) = \bot$, otherwise $v(e)$ if the value $\widehat{f}(v_1, \ldots, v_n)$.

If $v(e) = \bot$, we say that $v(e)$ is *undefined* under the value assignment v.

The next step is to define the semantics of formulas. To this end we will use the three-valued logic having the Boolean values `true` and `false` and the undefined value $\bot$. In this logic, for example, a disjunction is true if at least one of its members is true; false if all of them are false and undefined otherwise. We omit details due to lack of space.

Path Feasibility. Suppose that π is a path in a program using only assignments and tests. The *path feasibility problem* is the problem of finding initial values of variables which make the path feasible, i.e., values so that the program execution starting with this assignment satisfies all tests on the paths and raises no exceptions. The path feasibility problem can be reduced to the following *constraint satisfaction problem for $\mathcal{LL}$*:

Given a finite set $\{L_1, \ldots, L_n\}$ of literals, find a value assignment v which makes all of these literals true.

Evidently, the path feasibility problem can be reduced in a straightforward way to the constraint satisfaction problem for $\mathcal{LL}$. We do not give full details here but restrict ourselves to an example.

Example 2. Consider the path from Example 1. This path can be translated to the following constraint.

$i_1 = LastIndexOf_1(s_1, \text{'/'})$. $i_2 = Length(\texttt{"http://"})$.
$\neg i_1 < 0$. $s_3 = Substring_2(s_1, \ell, i_1 - i_2)$.
$s_2 = Substring_1(s_1, i_1 + 1)$. $StartsWith(s_3, \texttt{"www."})$.
$Contains(s_2, \texttt{"EasyChair"})$. $s_4 = Substring_1(s_3, Length(\texttt{"www."}))$.
$StartsWith(s_1, \texttt{"http://"})$. $s_4 = \texttt{"live.com"}$.

This constraint is satisfiable if and only if there exists an initial value of the variable s_1 so that the program will follow the intended path.

3   The Core String Language

In this section we will define the so-called *core string language $\mathcal{CL}$*. This language contains a smaller number of functions than $\mathcal{LL}$ but we will be interested in a larger fragment of this language, including propositional connectives and *bounded quantifiers* of a special form. This language will play the role of an intermediate language between the string library language and an SMT solver. We will start with defining $\mathcal{CL}$ and giving a translation of $\mathcal{LL}$ into $\mathcal{CL}$. This translation will also define the semantics of the $\mathcal{LL}$ functions in a strict way as opposed to a less formal description in Table 1.

Definition 1 (core string language). For simplicity we will usually say *the library language* instead of the string library language and *the core language* instead of the core string language. Likewise, we will simply say *library functions* and *core functions*. The core language contains the following functions and predicates:

1. The string functions *Length* and *Chars*. We will write $\ell(s)$ instead of $Length(s)$ and $s[i]$ instead of $Chars(s, i)$.
2. The standard functions and predicates of linear integer arithmetic.
3. Some functions and predicates on characters. We do not specify the set of all these functions but will use two of them (comparison $<$ and *ToUpper*) later.

The *formulas* of the language are defined as follows:

1. Every predicate specified above is a formula;
2. If F_1 and F_2 are formulas, then $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$ and $F_1 \rightarrow F_2$ are formulas.
3. If F is a formula, i a variable of the sort $\mathcal{I}$ and ℓ, u expressions of the sort $\mathcal{I}$ not containing i, then $(\forall i)(\ell \leq i \wedge i \leq u \rightarrow F)$ and $(\exists i)(\ell \leq i \wedge i \leq u \wedge F)$ are formulas. In the following, we will write these instead as: $(\forall i \in [\ell \ldots u])F$ and $(\exists i \in [\ell \ldots u])F$ respectively.

Note the following equivalences:

$$\neg(\forall i \in [\ell \ldots u])F \equiv (\exists i \in [\ell \ldots u])\neg F; \quad \neg(\exists i \in [\ell \ldots u])F \equiv (\forall i \in [\ell \ldots u])\neg F.$$

Table 2. Exception conditions for the library functions

Function	exception condition
$Chars(s, i)$	$i < 0 \vee i \geq \ell(s)$
$IndexOf_4(s_1, s_2, i)$	$i < 0 \vee \ell(s_1) \geq i$
$Replace_2(s_1, s_2, s_3)$	$\ell(s_2) = 0$
$Substring_1(s, i)$	$i < 0 \vee i > \ell(s)$
$Substring_2(s, i_1, i_2)$	$i_1 < 0 \vee i_2 < 0 \vee i_1 + i_2 > \ell(s)$

Let us now define formally the translation of the library language into the core language. We will do this by first defining exception conditions for all library functions and then the library functions themselves using the core language. The exception conditions are given in Table 2. Note that ℓ is the only non-arithmetical core function used in the exception conditions.

Let us introduce two abbreviations for the core language as follows.

$$s_1[i \ldots j] = s_2 \stackrel{\text{def}}{=} \ell(s_2) = j - i + 1 \wedge (\forall k \in [i \ldots j])(s_1[k] = s_2[k - i]). \quad (1)$$

$$s_1 \sqsubseteq s_2 \stackrel{\text{def}}{=} (\exists i \in [0 \ldots \ell(s_2) - \ell(s_1)])s_2[i \ldots i + \ell(s_1) - 1] = s_1. \quad (2)$$

One can easily show that $s_1 \sqsubseteq s_2$ is equivalent to $Contains(s_2, s_1)$ and to $\exists s_3 \exists s_4(s_3 \cdot s_1 \cdot s_4 = s_2)$. The difference is that $s_1 \sqsubseteq s_2$ is a formula of the core language.

Let us now give a definition for every library function in the core language. This, together with the definition of exception conditions, also provides a precise semantics for the library functions instead of the less formal explanation given in Table 1. The definitions of the library functions are given in Table 3.

We are interested in the following fragment of the core language.

Definition 2. Let F be a formula of either the library or the core language such that (i) the string variables occurring in F are $s_1, \ldots, s_n$ and (ii) F contains no free occurrences of integer variables. The formula F is said to be a *fixed-length formula* if it has the form

$$\ell(s_1) = i_1 \wedge \ldots \wedge \ell(s_n) = i_n \wedge F',$$

where $i_1, \ldots, i_n$ are concrete integer constants occurring in F'. The *fixed-length fragment* of the library (respectively, core) language is the set of all fixed-length formulas of this language.

Table 3. Definitions of some library predicates

function/predicate	definition
$Chars(s, i)$	$s[i]$
$Compare(s_1, s_2) = 0$	$s_1 = s_2$
$Compare(s_1, s_2) < 0$	$(\exists i \in [0 \dots \ell(s_1) - 1])(\ell(s_2) > i \wedge s_1[0 \dots i - 1] = s_2[0 \dots i - 1] \wedge$ $(\ell(s_1) = i \vee s_1[i] < s_2[i]))$
$Compare(s_1, s_2) > 0$	$(\exists i \in [0 \dots \ell(s_2) - 1])(\ell(s_1) > i \wedge s_1[0 \dots i - 1] = s_2[0 \dots i - 1] \wedge$ $(\ell(s_2) = i \vee s_2[i] < s_1[i]))$
$Concat(s_1, s_2)$	$s_1 \cdot s_2$
$Contains(s_1, s_2)$	$s_2 \sqsubseteq s_1$
$Equals(s_1, s_2)$	$s_1 = s_2$
$IndexOf_4(s_1, s_2, i_1) = i_0$	$(i_0 = -1 \wedge s_2 \not\sqsubseteq s_1[i_1 \dots \ell(s_1) - 1]) \vee$ $(i_0 \geq i_1 \wedge s_1[i_0 \dots i_0 + \ell(s_2) - 1] = s_2 \wedge$ $(\forall j \in [i_1 \dots i_0 - 1])(s_1[j \dots j + \ell(s_2) - 1] \neq s_2))$
$LastIndexOf_1(s, c) = i$	$(i = -1 \wedge (\forall j \in [0 \dots \ell(s) - 1])s[j] \neq c) \vee$ $(i \geq 0 \wedge s[i] = c \wedge (\forall j \in [i + 1 \dots \ell(s) - 1])s[j] \neq c)$
$Replace_2(s_1, s_2, s_3)$	no definition exists
$StartsWith(s_1, s_2)$	$(\exists i \in [0 \dots \ell(s_1) - \ell(s_2)])s[0 \dots i - 1] = s_2$
$Substring_1(s, i)$	$s[i \dots \ell(s) - 1]$
$Substring_2(s, i_1, i_2)$	$s[i_1 \dots i_1 + i_2 - 1]$
$ToString(c) = s$	$\ell(s) = 1 \wedge s[0] = c$
$ToUpper(s_1) = s_2$	$\ell(s_2) = \ell(s_1) \wedge (\forall i \in [0 \dots \ell(s_1) - 1])Upper(s_1[i], s_2[i])$

Theorem 1. *The satisfiability problem for the fixed-length fragment of the core language is decidable.*

Proof. Since $i_1, \dots, i_n$ are integer constants, every value assignment that satisfies the formula assigns to s_k a string of length i_k, for all $k = 1 \dots n$. There exists only a finite number of value assignments with this property, so by substituting these value assignments the formula can be replaced by a finite disjunction of formulas with no free variables. It remains to show that satisfiability of closed formulas of the core language is decidable. This can be proved by induction on the depth of quantifiers in a formula F. If the number of quantifiers in F is 0, F is a variable-free quantifier-free formula which can simply be evaluated. Suppose now that F has at least one quantifier, we will then show how to eliminate a quantifier in F. Take any quantified subformula G of F that is not in the scope of another quantifier. Without loss of generality we can assume that G is of the form $(\forall i \in [e_1 \dots e_2])H(i)$. Then e_1 and e_2 are variable free, so they can be evaluated to concrete integer values k_1 and k_2, hence G can be replaced by the conjunction $\bigwedge_{k_1 \leq k \leq k_2} H(k)$ having a smaller quantifier depth.

Note that this proof works for the string library with any finite number of functions or predicates on characters, since for such functions we only need that they can be computed. Section 6 describes how our implementation uses the theory of arrays to delay relying on finite domains for solving constraints.

As a consequence of this theorem we obtain the following result.

Theorem 2. *The satisfiability problem for the fixed-length fragment of the library language without the function Replace is decidable.*

Proof. Using definitions of Table 3 one can translate any formula of the library language without the function *Replace* into an equivalent formula of the core language. Since the translation does not introduce new free variables, any fixed-length formula is translated into a fixed-length formula. Then apply Theorem 1 on the decidability of the fixed-length fragment of the core language.

Theorem 2 and its proof provide a foundation for our method of constraint solving.

4   Library Language Decidability and Undecidability Results

In this section we consider the full (not fixed-length) library language. We will show that constraint solving for this language is undecidable. We will also point out that the decidability of a very large fragment of this language is equivalent to the decidability problem of a well-known problem related to word equations. Finally, we will prove some decidability results.

It is easy to see that several functions and predicates of the string library can be expressed using string concatenation. Among the representative subset selected by us these are *Concat*, *Contains*, *Equals* and *StartsWith*. Solving constraints using only such functions and predicates can be reduced to solving word equations and so is decidable. However, this fragment is hardly very practical.

Word Equations and Equal Length Constraints. Let us call the subset of $\mathcal{LL}$ without *Replace*, *ToUpper* the *pure library language*. It is interesting that path feasibility in the pure library language is equivalent to a well-known extension of word equations whose decidability is an open problem. This problem is known as *word equations with the equal length predicate*. The equal length predicate, denoted by $\ell\ell$ is true on a pair of strings s_1, s_2 if and only if s_1 and s_2 have the same length. The decidability of word equations with the equal length predicate is an open problem [2,12].

Theorem 3. *The path feasibility problem in the pure library language is decidable if and only if word equations with the equal length predicate are decidable.*

Proof. We will show how to reduce the two problems to each other. To this end, we will first reformulate the constraint satisfaction problem for $\mathcal{LL}$ as a problem on strings. To represent a character c as a string we will use the string consisting of this character. To represent non-negative integers, we will use the following idea. Let us fix a letter of the alphabet, for example, $|$. We will use the string $| \ldots |$ of length n, denoted by $\widehat{n}$ as a representation for the number n and call such strings *numerals*. To represent a negative number $-n$ we can use, for example, the string $- | \ldots |$ of length $n+1$, that is $-\widehat{n}$. Let us prove several facts about this representation. When we write that some relation can be represented we mean that it can be represented using an existential formula. Note the well-known fact [2] that inequations of words can be represented using equations, hence we will not consider negative literals in the proof.

Using word equations one can express that a string s is a numeral. Indeed, consider the equation $s| = |s$. The solutions of this equation are exactly all numerals. *Using*

word equations one can express the addition on integers. In our representation the addition on non-negative integers becomes string concatenation: indeed, $\widehat{mn} = \widehat{m+n}$ for all non-negative integers m, n. It is not hard to represent addition on all integers too. *One can express the equal length predicate using the length function and vice versa.* One direction is obvious since $\ell\ell(s_1, s_2) \equiv \ell(s_1) = \ell(s_2)$. In the other direction, note that the length of a string s is equal to n if and only if s and $\widehat{n}$ have equal length. What remains to note is that for the pure library language functions both the equality and the inequality between these functions can be represented using string concatenation and the length function.

Let us now prove an undecidability result.

Theorem 4. *The path feasibility problem for the library language is undecidable.*

Proof. For every character c of the alphabet consider the following function ℓ_c: for every string s, $\ell_c(s) = \widehat{n}$, where n is the number of occurrences of c in n. We will use the following result from [2]: the existential theory of words with concatenation, the equal length predicate and functions ℓ_0, ℓ_1 is undecidable. We already proved in Theorem 3 that the equal length predicate is expressible in the string library language, it remains to note that, for every character c, ℓ_c is expressible using the library function *Replace*.

One can prove other decidability results about automatic structures, we will only briefly sketch how one can prove them here. Let us call a predicate or a function on strings *automatic* if it can be represented by a finite automaton, for details see [10]. To apply this definition to integers and characters we assume that they are represented respectively as numerals and as one-character strings. The string library contains several automatic functions, namely *Chars*, *Compare*, *Equals*, *Length*, *StartsWith*, *ToString*, *ToUpper*. Other functions are not automatic but have automatic instances when one or more arguments are instantiated to constants. For example, for every constant string s_2, $Concat(s_1, s_2)$ considered as a function of s_1 is automatic. It is also automatic if we fix the first argument s_1 to be constant. It is known that the full first-order theory of every automatic structure (structure in which all predicates and functions are automatic) is decidable. However, this result is hardly interesting in practice for two reasons. Firstly, using automata-based method on large alphabets is prohibitively expensive. Secondly, the resulting decidable fragment is too narrow for applications, for example, it does not include concatenation and integer addition.

5 Solving Constraint Satisfaction Problems in the Library Language

Our algorithm for checking constraint satisfaction is described here and works as follows. First, we replace subterms by fresh variables to obtain a flattened constraint C. Then, produce a so-called *integer abstraction* of the problem. The integer abstraction is a quantifier-free formula I of linear arithmetic over two kinds of integer variables: those coming from the constraint C and those denoting the lengths of string variables occurring in C. After that we look for *small* solutions to I. Small solutions are obtained from

Table 4. Integer abstraction of library predicates

function	abstraction
$Compare(s_1, s_2) = c$	$(\ell(s_1) = 0 \rightarrow c \leq 0) \wedge (\ell(s_2) = 0 \rightarrow c \geq 0)$
$Concat(s_1, s_2) = s$	$\ell(s) = \ell(s_1) + \ell(s_2)$
$Contains(s_1, s_2)$	$\ell(s_1) \geq \ell(s_2)$
$IndexOf_4(s_1, s_2, i_1) = i$	$i = -1 \vee (i \geq i_1 \wedge i + \ell(s_2) \leq \ell(s_1))$
$LastIndexOf_1(s, c) = i$	$i = -1 \vee i < \ell(s)$
$Replace_2(s_1, s_2, s_3) = s_0$	$(\ell(s_2) \geq \ell(s_3) \rightarrow \ell(s_1) \geq \ell(s_0)) \wedge$ $(\ell(s_2) \leq \ell(s_3) \rightarrow \ell(s_1) \leq \ell(s_0))$
$StartsWith(s_1, s_2)$	$\ell(s_1) \geq \ell(s_2)$
$Substring_1(s_1, i) = s_0$	$\ell(s_0) = \ell(s_1) - i$
$Substring_2(s_1, i, j) = s_0$	$\ell(s_0) = j \wedge \ell(s_1) \geq i + j$
$ToString(c) = s$	$\ell(s) = 1$
$ToUpper(s_1) = s_0$	$\ell(s_0) = \ell(s_1)$

feasible solutions by imposing and repeatedly tightening bounds on string lengths. If there is no such solution, then the original constraint is unsatisfiable. If I has a solution it gives us the lengths of all string variables in C. When we fix the length, we obtain a fixed-length formula in the core language which can be decided by a finite domain solver. If this formula has no solution, backtrack and try to find another solution of I.

Flattening. A constraint C is called *flat* if all string library functions occur in C at the top level, that is, for every term of formula $p(t_1, \ldots, t_n)$ occurring in C, where p is different from equality, the terms $t_1, \ldots, t_n$ contain no occurrences of the string library functions. For example, the constraint $Substring_1(s_1, i_1 + i_2)$ is flat while the constraint $s_1 = Concat(s_2, Substring_1(s_3, 1))$ is not, since $Substring_1$ does not occur at the top level. Constraints with flat literals are created by introducing extra variables for subterms. For example, the constraint of Example 2 will become as shown on the right.

$i_1 = LastIndexOf_1(s_1, \text{'/'})$.
$\neg i_1 < 0$.
$s_2 = Substring_1(s_1, i_1 + 1)$.
$Contains(s_2, \texttt{"EasyChair"})$.
$StartsWith(s_1, \texttt{"http://"})$.
$i_2 = Length(\texttt{"http://"})$.
$s_3 = Substring_2(s_1, i_2, i_1 - i_2)$.
$StartsWith(s_3, \texttt{"www."})$.
$i_3 = Length(\texttt{"www."})$.
$s_4 = Substring_1(s_3, i_3)$.
$s_4 = \texttt{"live.com"}$

Integer abstraction. The integer abstraction of a literal defines necessary conditions for the literal to be true. This implies that every solution to the literal must also be a solution to the integer abstraction. For every literal L in a flat constraint its integer abstraction is built as follows. First, let F^e be the exception condition corresponding to the literal L in Table 2; F^e is `false` if there are no matching exception conditions for L. If the literal L matches an entry in Table 4 with formula F^i, then the integer abstraction of L is given as $\neg F^e \wedge F^i$. If L is a negation $\neg L'$ and L' matches an entry in Table 4, then the abstraction is given as $\neg F^e$; otherwise, the abstraction of L is set to L. The flat constraint for our example has the integer abstraction given in Figure 2.

$i_1 = LastIndexOf_1(s_1, '/')$	$i_1 = -1 \vee i_1 < \ell(s_1)$
$\neg i_1 < 0$	$\neg i_1 < 0$
$s_2 = Substring_1(s_1, i_1 + 1)$	$\neg(i_1 + 1 < 0 \vee i_1 + 1 > \ell(s_1)) \wedge \ell(s_2) = \ell(s_1) - i_1 - 1$
$Contains(s_2, \texttt{"EasyChair"})$	$\ell(s_2) \geq 9$
$StartsWith(s_1, \texttt{"http://"})$	$\ell(s_1) \geq 7$
$i_2 = Length(\texttt{"http://"})$	$i_2 = 7$
$s_3 = Substring2(s_1, i_2, i_1 - i_2)$	$\neg(i_2 < 0 \vee i_1 - i_2 < 0 \vee i_2 + i_1 - i_2 > \ell(s_1)) \wedge$ $\ell(s_3) = i_1 - i_2 \wedge \ell(s_1) \geq i_2 + i_1 - i_2$
$StartsWith(s_3, \texttt{"www."})$	$\ell(s_3) \geq 4$
$i_3 = Length(\texttt{"www."})$	$i_3 = 4$
$s_4 = Substring_1(s_3, i_3)$	$\neg(i_3 < 0 \vee i_3 > \ell(s_3)) \wedge \ell(s_4) = \ell(s_3) - i_3$
$s_4 = \texttt{"live.com"}$	$\ell(s_4) = 8$

Fig. 2. Integer abstraction of the example program

Fixed-length constraint satisfaction problems. A constraint C is said to be *fixed-length* if for every string variable s occurring in C, the constraint also contains a literal of the form $\ell(s) = i$, where i is an integer constant. One can easily note that in this case for every solution of C the length of s is i.

Consider any flat constraint C and its integer abstraction I. If I is unsolvable, then C has no solution. Let us now take any value assignment v that solves I and consider the constraint C' obtained by adding to C all constraints of the form $\ell(s) = v(i)$, where s is a string variable occurring in C and i the integer variable denoting the length of s. One can immediately see that C' is a fixed-length constraint and that every solution to C' is also a solution to C. Our next observation is that the satisfiability problem for fixed-length constraints is decidable by Theorem 1.

6   Implementation in Pex and Integration with the SMT Solver Z3

In this Section we describe how string library functions and symbolic string constraints are represented. We then describe how the Pex [16] tool produces symbolic string constraints. We observe that several of the string library functions can be handled using two functions we call Shift and Fuse. Finally, we describe using the SMT solver Z3 for solving string constraints. We experiment with different strategies for the integration.

In Pex, Strings are represented as an abstract type String. We use one predicate and two functions for accessing strings. The predicate $\texttt{null}(s)$ is true if the string s is a null pointer. The function $\texttt{length}(s)$ results in the length of s, and the function $\texttt{chars}(s)$ results in an array, whose domain consists of 32-bit bit-vectors and the range is the set of unicode characters (16-bit bit-vectors).

Building abstract execution paths. All string library functions have implementations in Microsoft's .NET base class library, but many of these are in native code and therefore not in the scope of what Pex can analyze (Pex only analyzes .NET code). Pex therefore contains straightforward implementations of each of the string library functions written in C#. We show the implementation of the IndexOf function as an example below.

```
public static int IndexOf(string text, string key, int start, int count)
{
    if (text == null) throw new NullReferenceException();
    if (key == null) throw new ArgumentNullException();
    if (start < 0 | count < 0 | start + count < 0 |
        start + count > text.Length)
        throw new ArgumentOutOfRangeException();
    if (key.Length == 0) return start;
    if (count < key.Length) return -1;
    return IndexOfC(text, key, start, count);
}


private static int IndexOfC(string text, string key, int start, int count)
{
    int end = start + count - key.Length + 1;
    for (int i = start; i < end; i++) {
        bool b = true;
        for (int j = 0; b && j < count; j++)
            b &= text[i + j] == key[j];
        if (b) return i;
    }
    return -1;
}
```

The implementation contains two parts, the preamble within the body of IndexOf is a straight-line code sequence that checks for exception conditions and boundary values. These checks include the conditions listed in Table 2 and parts from the abstraction of Table 4 that cover also the concrete case. Then, the portion of the string function that we wish to abstract is encoded in IndexOfC.

At this point we can run standard dynamic symbolic execution with the string library functions by using the instructions within IndexOf and IndexOfC for both the concrete and symbolic execution. Our aim is however to abstract the part of IndexOfC into core string constraints. For this purpose we introduce the uninterpreted function IndexOfA. When executing IndexOfC in dynamic symbolic execution, we do not add constraints to the path condition, but set the symbolic result to IndexOfA($text, key, start, count$).

Functions, such as Concat and Substring, do not depend on the string tokens. Pex encodes such functions using two primitives, Shift and Fuse, axiomatized by $\mathsf{Shift}(a, i)[j] \simeq a[i + j]$ and $\mathsf{Fuse}(a, i, b)[j] \simeq \mathbf{if}\ j < i\ \mathbf{then}\ a[j]\ \mathbf{else}\ b[j]$. We can then replace $\mathsf{chars}(\mathsf{ConcatA}(a, b))$ by $\mathsf{Fuse}(\mathsf{chars}(a), \mathsf{length}(a), \mathsf{Shift}(\mathsf{chars}(b), -\mathsf{length}(a)))$, and $\mathsf{chars}(\mathsf{SubstringA}(a, i, j))$ by $\mathsf{Shift}(\mathsf{chars}(a), i)$.

Solving string constraints. For each execution path we get a path condition and perform multiple queries to Z3 using the algorithm outlined next. It uses constants $\mathcal{U}$ to bound the length of strings, $\mathcal{N}$ to bound the number of calls to Z3. In our implementation, both $\mathcal{U}$ and $\mathcal{N}$ are used to bound the search.

Phase 1. Assert the path condition π, the axioms for Fuse and Shift, and the axiom $\forall s\ .\ 0 \leq \mathsf{length}(s) < \mathcal{U}$, where $\mathcal{U}$ is a fresh constant. If the constraints are unsatisfiable, then fail; otherwise enter the second phase.

Phase 2. We are given a path constraint π that is satisfiable with respect to partial unfoldings of the supplied axioms. Find the smallest power of two for $\mathcal{U}$ such that the constraints have a model. Set $\mathcal{N} \leftarrow 0$.

   1. Extract values from the model that suffice to create a finite unfolding of all quantifiers used in Table 3 for the functions: IndexOf, LastIndexOf,

Contains, Compare, and Equals. Thus, we assert the definitions of these functions replacing all quantifiers of the form $(\forall i \in [a \dots b])\varphi(i)$ with a finite set of assertions $\varphi(v(a)), \dots, \varphi(v(b))$.

2. Instantiate the definitions of Shift and Fuse[1].
3. If the constraints are satisfiable, return the current model.
4. Fail if $\mathcal{N}$ or $\mathcal{U}$ exceed pre-configured bounds.
5. Otherwise, undo the assertions from steps 1 and 2 and force a solution with increased lengths by asserting $\Sigma_i \ell(s_i) > \Sigma_i v(\ell(s_i))$ where $\ell(s_i)$ occur in π.
6. If the new constraints are unsatisfiable, then fail.
7. Otherwise, repeat step 1 with $\mathcal{U} \leftarrow 2 \cdot \mathcal{U}, \mathcal{N} \leftarrow \mathcal{N} + 1$,

Note that step 5 can prevent exploring models where string lengths add up to the previous value, but are re-distributed in a different way. While $\mathcal{N}$ and $\mathcal{U}$ impose limits on which models are explored, it is the case that our implementation in Pex makes implicit use of properties of the bit-vector solver in Z3 in the folowing way: Z3 has a preference for models of bit-vectors where the most significant bits are set, so the progression of lengths tends to grow in proportion to $\mathcal{U}$.

To distribute length increases fairly among strings, Pex furthermore excludes strings whose length was increased recently, unless this exclusion would cause the constraints to become unsatisfiable.

7 Experiments

We applied Pex on the EasyChair[2] query given in Figure 1 using different search strategies to flip branches of already discovered execution paths (which includes unrolling loops): breadth-first (BFS), depth-first (DFS), flipping of Random branches, and Pex' default strategy [17] (which combines several heuristics). We compare those strategies with a *partial* implementation of the algorithm described above, where only Concat, Substring, Remove and Insert are abstracted, and the fully *abstract* version of the algorithm. We set the bounds of $\mathcal{U}$ at 2^{12}, and $\mathcal{N}$ at 3. Table 5 shows the re-

Table 5. Evaluation of string solver on EasyChair

mode	time/s	paths
BFS	7.90	214
DFS	3.65	51
Random	8.73	196
Default	0.83	35
Partial	0.61	30
Abstract	1.02	19

sults: BFS, DFS and Random search performed clearly worse than the default and abstract strategies. The partial abstraction resulted in the fastest run-time and fewer explored paths than the default exploration strategy, while complete abstraction required exploring fewer paths, but took slightly more overall time.

A different example that highlights the effectiveness of the abstraction can be constructed by creating a program test with a conjunction of the form $s.\texttt{IndexOf}(s_1)! = -1\&\&s.\texttt{IndexOf}(s_2)! = -1\&\&\dots$, where $s_1, s_2, \dots, s_c$ are different non-overlapping strings with a long common prefix. The goal is to synthesize a string s that contains all the substrings $s_1, s_2, \dots s_c$. Table 6 summarizes the results of trying a progression of

---

[1] While necessary for completeness, this step has not had any effect in our experiments.

[2] All experiments were performed with a Intel Core 2 CPU T7400 @ 2.16 Ghz, 4GB RAM.

Table 6. Evaluation of string solver on IndexOf progression

c	1		2		3		4		5		6	
mode	time	paths	time	paths	time	paths	time	paths	time	paths	time	paths
BFS	0.14	5	4.50	140	73.34	1432	960.08	7727	timeout		timeout	
Random	0.26	7	0.42	9	4.93	57	16.70	108	40.55	199	154.36	541
Default	0.23	6	0.62	14	7.03	78	7.76	80	8.81	82	193.49	637
Abstract	0.15	6	0.33	8	0.99	10	2.02	12	4.44	14	6.67	16

$c = 1, \ldots, 6$ such conjuncts. DFS search, not shown, does not even manage to explore a single path, other strategies are also inferior to abstraction.

8   Conclusion

We presented a two-tier approach for generating finite models of path constraints for string-manipulating programs. Our approach views constraints from string libraries as extensions of the word equation problem and we identified decidable, undecidable, and open problems in the context of word equations. Our approach was integrated with the dynamic symbolic execution engine Pex.

Related Work. In the context of symbolic execution of programs, string abstractions has been recently studied by Ruan et.al. [13], where an approach based on a first-order encoding of string functions is proposed. They study C programs where strings are zero-terminated arrays whose lengths are bounded by constants. The first-order quantifiers can therefore be finitely unfolded and decided using a solver for linear arithmetic and assignments. Ruan et.al. [5] also fix length of strings in order to obtain a decidable fragment, but don't consider increasing the size of strings in the search of models. Shannon et.al. [15] use automata-based representations for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the java.lang.String class, and some other related classes. They integrate a numeric constraint solver, but apparently in a partial way. For example, string methods which return integers, such as IndexOf, cause case-splits over all possible return values within certain bounds. Automata-based methods have been pursued in the context of static analysis by Christensen et.al. [4], where automata, using the Mohri-Nederhof algorithm, represent over-approximations of possible string values. A motivation for the work is *SQL injection attacks*. The same motivation also inspired Fu et.al. [6]. They first solve Boolean and integer constraints to obtain a model and then proceed to solving string constraints by using the obtained model to build automata for the string constraints. To our knowledge, the mentioned automata-based methods require case analysis outside of their calls to their constraint solvers and automaton construction phases. In our framework, case analysis is integrated with the constraint solver pass.

Future work. A plethora of future work is possible in the context of exploring string manipulating programs. *Regular expressions* are often used by the discriminating programmer to accomplish string manipulation. In particular, our running EasyChair example can be directly encoded using a regular expression. But real regular expression

libraries can encode side-effects and non-regular properties. Can such extensions be handled by methods presented here? A different direction is to extend array property fragments [1,9] to handle common string queries. We would also like to use information encoded in the core language to control the constraint solver programmatically. For example, one could alternate quantifier unfolding with solving for the bounds.

References

1. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
2. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation. Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik (1988)
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS, pp. 322–335. ACM Press, New York (2006)
4. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
5. Dong, Y., Quan, Q., Zhang, J.: Priority-based energy aware and coverage preserving routing for wireless sensor network. In: VTC Spring, pp. 138–142. IEEE, Los Alamitos (2008)
6. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. In: COMPSAC, pp. 87–96 (2007)
7. Godefroid, P.: Compositional dynamic test generation. In: Proc. of POPL 2007, pp. 47–54. ACM Press, New York (2007)
8. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Notices 40(6), 213–223 (2005)
9. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 474–489. Springer, Heidelberg (2008)
10. Khoussainov, B., Nies, A., Rubin, S., Stephan, F.: Automatic structures: Richness and limitations. In: LICS, pp. 44–53 (2004)
11. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
12. Matiyasevich, Y.: Word Equations, Fibonacci Numbers, and Hilbert's Tenth problem. In: Workshop on Fibonacci Words, vol. 43, pp. 36–39 (2007)
13. Ruan, H., Zhang, J., Yan, J.: Test Data Generation for C Programs with String-Handling Functions. Theoretical Aspects of Software Engineering 0, 219–226 (2008)
14. Sen, K., Agha, G.A.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
15. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Taicpart-Mutation, Washington, DC, USA, pp. 13–22 (2007)
16. Tillmann, N., de Halleux, J.: Pex - white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
17. Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. Technical Report MSR-TR-2008-123, Microsoft (2008)

Symbolic String Verification: Combining String Analysis and Size Analysis[★]

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra

Department of Computer Science
University of California, Santa Barbara, CA, USA
{yuf,bultan,ibarra}@cs.ucsb.edu

Abstract. We present an automata-based approach for symbolic verification of systems with unbounded string and integer variables. Particularly, we are interested in automatically discovering the relationships among the string and integer variables. The lengths of the strings in a regular language form a semilinear set. We present a novel construction for length automata that accept the unary or binary representations of the lengths of the strings in a regular language. These length automata can be integrated with an arithmetic automaton that recognizes the valuations of the integer variables at a program point. We propose a static analysis technique that uses these automata in a forward fixpoint computation with widening and is able to catch relationships among the lengths of the string variables and the values of the integer variables. This composite string and integer analysis enables us to verify properties that cannot be verified using string analysis or size analysis alone.

1   Introduction

Static analysis of strings in programs have been an active research area with the goal of finding and eliminating security vulnerabilities caused by misuse of string variables. There have been two separate branches of research in this area: 1) *String analysis* that focuses on statically identifying all possible values of a string expression at a program point in order to eliminate vulnerabilities such as SQL injection and cross-site scripting (XSS) attacks [1, 4, 14, 16], and 2) *Size analysis* that focuses on statically identifying all possible lengths of a string expression at a program point in order to eliminate buffer overflow errors [5, 7, 12]. In this paper we present an automata based composite symbolic verification technique that combines these two analyses with the goal of improving the precision of both. We use a forward fixpoint computation to compute the possible values of string and integer variables and to discover the relationships among the lengths of the string variables and integer variables.

Similar to prior size analysis techniques [5, 7, 12] we associate each string variable with an auxiliary integer variable that represents its length. At each program point, we symbolically compute all possible values of all integer variables (including the auxiliary variables), as well as all possible values of all string variables. The reachable values of all integer variables are over-approximated as a Presburger arithmetic (linear arithmetic)

formula and symbolically encoded as *arithmetic automata* [2, 13]. Similar to some prior string analysis techniques [1, 16], the values that string variables can take are over-approximated as regular languages and symbolically encoded as *string automata*. Our composite analysis is as a forward fixpoint computation with widening on these arithmetic and string automata.

There are two challenges we need to overcome to connect the information contained in the string automata and the arithmetic automata (hence, improving the precision of both) during our composite analysis: 1) Given a string automaton, we need to derive the arithmetic automaton that accepts the length of the language accepted by the string automaton, and 2) Given an arithmetic automaton, we need to restrict a string automaton so that the length of the language is accepted by the arithmetic automaton.

To tackle the first challenge, we present techniques for constructing a *length automata* for a given regular language. It is known that the length of the language accepted by a DFA forms a semilinear set. Given an arbitrary DFA, we are able to construct DFAs that accept either unary or binary representation of the length of its accepted words. The unary automaton can be used to identify the coefficients of the semilinear set, while the binary automaton can be composed with other arithmetic automata on integer variables to enforce or check length constraints.

To tackle the second challenge, we identify the boundary of the lengths of string variables from the arithmetic automaton. Precisely, we compute the lower and upper bound of the values of the string lengths accepted by the arithmetic automaton. We prove that, given a one-track arithmetic automaton, the lower bound forms a shortest path to an accepting state while the upper bound (if it exists) forms the longest loop-free path. Both can be computed in linear complexity to the size of the arithmetic automaton. We can restrict the target string automaton by intersecting the string automaton that accepts arbitrary strings within this boundary.

Finally, the performance of our analysis relies on efficient automata manipulation. We implement our analysis using a symbolic automata representation (MBDD representation from the MONA automata package) and leverage efficient manipulations on MBDDs, e.g., determinization and minimization.

Motivating Examples. Below, we present two motivating examples to demonstrate the advantages of the composite string and size analysis technique proposed in this paper. Consider a *PHP segment* that secures an identified vulnerable point [16] at line 218 in `trans.php`, distributed with `MyEasyMarket-4.1`.

```
1: <?php $www = $_GET["www"];
2:     $l_otherinfo = "URL";
3:     $www = ereg_replace("[^A-Za-z0-9 .\-@://]","",$www);
4:     if(strlen($www)<$limit)
5:        echo "<td>" . $l_otherinfo . ": " . $www . "</td>"; ?>
```

Without proper sanitization (lines 3 and 4) of the user-controlled variable $www, an attacker can inject the string <scriptsrc=http://evil.com/attack.js> and perform a XSS attack at line 5. Above code prevents such attacks by: (1) removing abnormal characters from $www at line 3, and (2) limiting the length of $www at line 4. Our analysis shows that this code segment is free from attacks by showing that at line 5 (1)

the length of the string $www is less than the allowed limit, and (2) under that limit the string variable $www cannot contain a value that matches the attack pattern. Note that if one performs solely size analysis, without knowing the contents of $www, the length of $www can not be determined precisely after line 3. On the other hand, if one performs solely string analysis, the branch condition at line 4 must be ignored. Both of these approximations may lead to false alarms.

Now, consider a standard strlen routine in C that returns the length of a given string by traversing each character until hitting the end character, i.e., '\0'. This kind of standard string routines are widely used in legacy C systems, e.g., Apache, Samba, Sendmail, and WuFTP.

```
unsinged int strlen(char *s){
1:     char *ptr = s;
2:     unsigned int cnt =0;
3:     while(*ptr != '\0'){
4:          ++ptr;
5:          ++cnt;
6:     }
7:     return cnt;   }
```

Let *s.*length* denote the size of the string pointed by s. An essential property of this routine is that at line 7, cnt = *s.*length*, which can be used as the summary of this routine and significantly alleviates size analysis overhead [5, 15], however, none of the size analysis tools prove this property before using it. Our composite analysis is capable of proving this property. We first construct an assertion (arithmetic) automaton that accepts the values that satisfy cnt = *s.*length*. We then conduct our composite analysis by computing the forward fixpoint with widening. Upon reaching the fixpoint, at line 7, (1) the arithmetic automaton actually catches the relation that $*s.length = *ptr.length + cnt$, and (2) the string automaton of *ptr only accepts $\{\epsilon\}$. We prove the property by showing that the intersection of the language of (1) and the length of the language of (2) is included in the language of the assertion automaton.

In addition to earlier work on string analysis [1, 4, 14, 16] and size analysis [5, 7, 12] that motivated our work, there has been some recent work on analyzing string and integer variables together during symbolic execution [6, 11, 15]. Unlike our approach, these are unsound techniques targeted towards testing and they do not try to compute an over-approximation of the reachable states via widening. Hence, they cannot prove properties of above program segments. Compared to [8, 9] that use abstract interpretation for reasoning relational properties among the contents of symbolic intervals of arrays, our analysis traverses concrete values of string and integer variables using automata and addresses language properties.

This paper is organized as follows. We present the length automata construction in Section 2. We present our composite analysis technique that integrates string and arithmetic analyses in Section 3. We present our experiments with our prototype tool in verifying small C routines, buffer-overflow benchmarks and PHP web applications in Section 4. We conclude the paper in Section 5.

2   Length Automata Construction

Given a string automaton M, we want to construct a DFA M_b (over a binary alphabet) such that $L(M_b)$ is the set of binary representations of the lengths of the words accepted by M. We tackle this problem in two steps. We first construct a DFA M_u (over a unary alphabet) such that $L(M_u)$ is the set of unary representations of the lengths of the words accepted by M. It is known that this set is a semilinear set. We identify the formula that represents the semilinear set from M_u. We then construct M_b from the formula, such that $w \in L(M_b)$ if and only if the binary value of w satisfies the formula. I.e., the unary representation of the binary value of w is in $L(M_u)$.

A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$ where Q is a finite set of states, q_0 is the initial state, Σ is a finite set of symbols. $F : Q \to \{-, +\}$ is a mapping function from a state to its status. Given a state $q \in Q$, q is an accepting state if $F(q) = +$. $\delta : Q \times \Sigma \to Q$ is the transition function. The cardinality of a finite set A is denoted as $\sharp A$. The set of arbitrary words over a finite alphabet Σ is denoted as Σ^*. The length of a word $w \in \Sigma^*$ is denoted as $|w|$. A state q of M is a *sink* state if $\forall \alpha \in \Sigma, \delta(q, \alpha) = q$ and $F(q) = -$. In the following sections, we assume that for all unspecified pairs (q, α), $\delta(q, \alpha)$ goes to a *sink* state. In the constructions below, we also ignore the transitions that lead to a sink state.

A string automaton M is a DFA that consists of a tuple of $\langle Q, q_0, B^k, \delta, F \rangle$. M accepts a set of words, where each symbol is encoded as a k-bit string.

Length Constraints on String Automata.   We are interested in characterizing lengths of the accepted words. We characterize these lengths as a set of natural numbers by a *length constraint*. Formally speaking, the length constraint of a given string automaton M is a formula f over a variable x, such that $f[c/x]$ evaluates to true if and only if there exists a word w, such that $w \in L(M)$ and $c = |w|$.

Property 1: For any DFA M, $\{|w| \mid w \in L(M)\}$ forms a semilinear set.

Property 2: For any DFA M, f_M is in the form that $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = a_j + b_j \times k$, where a_j, b_j and c_i are constants. f_M can be written as $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$, such that c_i, r_j, C, R are constants, and $\forall i, c_i < C$, and $\forall j, r_j < R$. We say that a semilinear set in this form is *well-formed*.

In the following, we give the algorithm to construct the automata that accept unary or binary representation of the length of the language accepted by a given string automata. This construction shows that the length constraint of a DFA is a well formed semilinear set, and hence gives a constructive proof of Property 1 and Property 2.

From String Automata to Unary Length Automata.   It is known that the unary representation of the values of a semilinear set can be uniquely identified by a unary automaton. In the following, we first show how to construct an automaton M_u (over a unary alphabet) from a given string automaton M, such that $L(M_u)$ is the set of unary representations of $\{|w| \mid w \in L(M)\}$. We say M_u is the unary length automaton of M.

Given a string automaton $M = \langle Q, q_0, B^k, \delta, F \rangle$, a naive construction of the unary length automaton is $M_u = \langle Q, q_0, B^1, \delta', F \rangle$, where $\delta'(q, 1) = q'$ if $\exists \alpha, \delta(q, \alpha) = q'$. However, M_u constructed this way will be an NFA. The MBDD representations that we

use cannot encode NFAs. Instead, we use a construction which combines the projection and determinization steps as follows.

Given a string automaton $M = \langle Q, q_0, B^k, \delta, F \rangle$, we first construct an intermediate automaton $M' = \langle Q, q_0, B^{k+1}, \delta', F \rangle$, where

- $\forall q, q' \in Q$, and both are not sink states, $\delta'(q, \alpha 1) = q'$, if $\delta(q, \alpha) = q'$.

M' is a DFA that accepts the same words as M except that each symbol in the word is appended with '1'. M_u can then be constructed from M' by projecting the first k bits away. This projection is done by iterative determinization and minimization. During determinization, the subset construction is applied on the fly.

From Unary Length Automata to Semilinear Set. Here we describe how to identify the well formed formula of a semilinear set from a unary automaton.

Property 3: A finite deterministic unary automaton $M = \langle Q, q_0, B^0, \delta, F \rangle$ can be in two forms: a linear list of states that starts from the initial state with finite length $\sharp Q$, or a linear list of states that starts from the initial state with finite length, C, and ends in a cycle with finite length, R, where $C + R = \sharp Q$ (i.e., a lasso).

Given a deterministic unary automaton, Q can be labeled such that

- $\sharp Q = n + 1$.
- $\forall 0 \leq i < n, \delta(q_i, 1) = q_{i+1}$.

Cycle Case: If $\exists 0 \leq m < n, \delta(q_n, 1) = q_m$, the well-formed formula of a unary automaton is $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$, where

- $C = m, R = n - m$.
- $\forall i, \exists q_t, t < m, F(q_t) = +, c_i = t$.
- $\forall j, \exists q_t, t \geq m, F(q_t) = +, r_j = t - m$.

No Cycle Case: Otherwise, the well-formed formula of a unary automaton is $\bigvee_i x = c_i$, where $\forall i, \exists q_t, t \leq n, F(q_t) = +, c_i = t$.

From Semilinear Set to Binary Length Automata. We propose a novel construction to derive a DFA M such that $L(M)$ is equal to the set of binary representations (from the least significant bit) of a well-formed semilinear set. We say M is a binary length automaton of the string automaton, the length of whose accepted words forms the semilinear set.

Assume that we are given a well-formed semilinear set $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$. Let N be $max(C, R)$. A DFA M that accepts the binary representation of the given semilinear set can be constructed as a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$, where:

- We assume that there exists a sink state $q_{sink} \in Q$, s.t., $F(q_{sink}) = -, \delta(q_{sink}, 0) = q_{sink}$ and $\delta(q_{sink}, 1) = q_{sink}$, and all transitions that are ignored in this construction are going to q_{sink}.
- Other than the sink state, each state $q \in Q$ is a tuple (t, v, b), where $t \in \{\mathtt{val}, \mathtt{rem}_t, \mathtt{rem}_f\}$, $v \in \{0, \ldots, N\}$, and $b \in \{\bot\} \cup \{1, \ldots, N\}$. $q.t$ is the type of state q, which indicates the meaning of the value of $q.v$ and $q.b$. While $q.t = \mathtt{val}$, $q.v$ is equal to

the value of the binary word accepted from the initial state to the current state, and $q.b$ is equal to the binary value of the previous bit in the word. We assume $2 \perp = 1$. While $q.t = \text{rem}_t$ or rem_f, $q.v$ is equal to the remainder of which the dividend is the value of the binary word accepted from the initial state to the current state and the divisor is R; $q.b$ is the remainder of which the dividend is the binary value of the previous bit in the accepted word and the divisor is R. $q.t = \text{rem}_t$ indicates the value of the binary word accepted from the initial state to the current state is greater or equal to C; $q.t = \text{rem}_f$ indicates the value is less than C.

- q_0 is $(\text{val}, 0, \perp)$.
- $\Sigma = \{0, 1\}$, i.e., B^1.
- $\delta(q, 1) = q'$ if and only if one of the following condition holds:
    - $q.t = \text{val}$, $q.v + 2q.b \geq C$, $q'.t = \text{rem}_t$, $q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
    - $q.t = \text{val}$, $q.v + 2q.b < C$, $q'.t = \text{val}$, $q'.v = q.v + 2q.b$, $q'.b = 2q.b$.
    - $q.t = \text{rem}_t$, $q'.t = \text{rem}_t$, $q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
    - $q.t = \text{rem}_f$, $q'.t = \text{rem}_t$, $q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
- $\delta(q, 0) = q'$ if and only if one of the following condition holds:
    - $q.t = \text{val}$, $q.v + 2q.b \geq C$, $q'.t = \text{rem}_f$, $q'.v = q.v \mod R$, $q'.b = (2q.b) \mod R$.
    - $q.t = \text{val}$, $q.v + 2q.b < C$, $q'.t = \text{val}$, $q'.v = q.v$, $q'.b = 2q.b$.
    - $q.t = \text{rem}_t$, $q'.t = \text{rem}_t$, $q'.v = q.v$, $q'.b = (2q.b) \mod R$.
    - $q.t = \text{rem}_f$, $q'.t = \text{rem}_f$, $q'.v = q.v$, $q'.b = (2q.b) \mod R$.
- $F(q) = +$, for all $q \in \{q \mid q.t = \text{val}, \exists i, q.v = c_i\} \cup \{q \mid q.t = \text{rem}_t, \exists j, q.v = (C + r_j) \mod R\}$; $F(q) = -$, o.w.

By definition, $\sharp Q$ is $O(N^2)$. Precisely, in our construction, the number of states that $q.t = \text{val}$ is bounded by C. The number of states that $q.t = \text{rem}_t$ is bounded by R^2 and the number of states that $q.t = \text{rem}_f$ is bounded by $C \times R$. On the other hand, we have observed that after minimization, $\sharp Q$ is often reduced to N.

An Incremental Algorithm. Below we give an incremental algorithm to construct a Binary Length Automaton (BLA) M. The construction is achieved by calling the procedure CONSTRUCT_BLA. The input is given as a well-formed semilinear formula, $\bigvee_{0 \leq i < n} x = c_i \vee \bigvee_{0 \leq j \leq m} \exists k.x = C + r_j + R \times k$. At line 3, we first build Q^b, the set of binary states that will be reached by calling the procedure ADD_BSTATE. A binary state is actually the value of the tuple (t, v, b) as described in the previous section. Each binary state is further associated with an index, a true branch and a false branch, which are used to construct the state graph. Briefly, ADD_BSTATE is a recursive function which incrementally adds the reached binary state if it has never been explored. Initially, the binary state is $(\text{val}, 0, \perp)$. Note that ADD_BSTATE is guaranteed to terminate since the number of binary states are bounded. Upon termination, all reached binary states will have been added to Q^b. For each binary state in Q^b, as line 4 to 9, we iteratively generate a state q and set its transition relation and accepting status, which are used to construct the final automaton at line 10.

We have implemented the above algorithms using the MONA DFA package. Minimal unary and binary length automata for a regular language are shown Figure 1 where the set recognized by these automata are $\{7 + 5k \mid k \geq 0\}$.

Algorithm 1. ADD_BSTATE(Q, C, R, t, v, b)

1: **if** $\exists q = (t, v, b) \in Q$ **then**
2:     **return** $q.index$;
3: **else**
4:     Create $q = (t, v, b)$;
5:     $q.index = \sharp Q$;
6:     $q.true = -1$;
7:     $q.false = -1$;
8:     Add q to Q;
9:     **if** $t == \mathtt{val} \wedge (v + 2 \times b \geq$ C) **then**
10:        $q.true =$ADD_BSTATE(Q, C, R, $\mathtt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
11:        $q.false =$ADD_BSTATE(Q, C, R, $\mathtt{rem}_f$, $v\%$R, $(2 \times b)\%$R);
12:     **else if** $t == \mathtt{val} \wedge (v + 2 \times b <$ C) **then**
13:        $q.true =$ADD_BSTATE(Q, C, R, $\mathtt{val}$, $v + 2 \times b$, $2 \times b$);
14:        $q.false =$ADD_BSTATE(Q, C, R, $\mathtt{val}$, v, $2 \times b$);
15:     **else if** $t == \mathtt{rem}_t$ **then**
16:        $q.true =$ADD_BSTATE(Q, C, R, $\mathtt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
17:        $q.false =$ADD_BSTATE(Q, C, R, $\mathtt{rem}_t$, $v\%$R, $(2 \times b)\%$R);
18:     **else if** $t == \mathtt{rem}_f$ **then**
19:        $q.true =$ADD_BSTATE(Q, C, R, $\mathtt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
20:        $q.false =$ADD_BSTATE(Q, C, R, $\mathtt{rem}_f$, $v\%$R, $(2 \times b)\%$R);
21:     **end if**
22:     **return** $q.index$;
23: **end if**

Algorithm 2. CONSTRUCT_BLA(C, R, $\mathcal{C} = \{c_1, c_2, \ldots c_n\}$, $\mathcal{R} = \{r_1, r_2, \ldots r_m\}$)

1: $Q^b = \emptyset$;
2: $Q = \emptyset$;
3: $init =$ADD_BSTATE(Q^b, C, R, $\mathtt{val}$, $0, \perp$);
4: **for** each $q^b \in Q^b$ **do**
5:     Add $q = q_{q.index}$ to Q;
6:     $\delta(q, 1) = (q^b.true \neq -1?q_{q^b.true} : q_{sink})$;
7:     $\delta(q, 0) = (q^b.false \neq -1?q_{q^b.false} : q_{sink})$;
8:     $F(q) = ((q^b.t == 0 \wedge \exists c \in \mathcal{C}.q^b.v == c) \vee (q^b.t == 1 \wedge \exists r \in \mathcal{R}.q^b.v ==$ $(r+$C$)\%$R$) :' +'?' -')$;
9: **end for**
10: Construct $M = \langle Q \cup \{q_{sink}\}, q_{init}, B^1, \delta, F \rangle$;

---

3   Composite Verification

We first introduce a simple imperative language (the syntax is similar to the one used in [15]) as our target language. This language consists of a set of labeled statements $l : stat$. Labels correspond to instruction addresses. We use s to denote a string variable, i to denote an integer variable, and c to denote a constant. Each $s \in S$ is associated
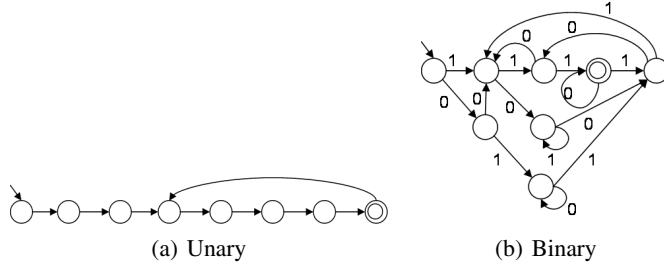
(a) Unary   (b) Binary

Fig. 1. The Length Automata of $(baaab)^+ab$

with one auxiliary integer variable, denoted as $s.length$. Let S denote the set of string variables and I denote the set of integer variables, and I_L denote the set of auxiliary variables. A statement can be one of the following:

- A termination statement `halt` or `abort`.
- A string assignment statement $s := strexp$, where $strexp$ is a string expression that can be one of the following:
  - `input`(i) which returns an arbitrary string value up to the length equal to the value of i.
  - a string variable $s \in S$.
  - a regular expression $regexp$ over S.
  - `prefix`(s, i) which returns the prefix of s up to the first c characters where c is equal to the value of i.
  - `suffix`(s, i) which returns the the suffix of s starting from the cth character, where c is equal to the value of i.
  - `concat`(s_1, s_2) that returns the concatenation of the value of s_1 and the value of s_2.
  - `replace`(s_1, s_2, s_3) that returns the result of the following actions: (1) scan the value of s_1 and find the substrings that match to the value of s_2, and (2) replace the matched substrings with the value of s_3.
- An integer assignment statement $i := intexp$, where $intexp$ is an integer expression in the form $\sum_t c_t * i_t$ that returns a value of the linear function $\sum_t c_t * i_t$, where each variable $i_t \in I \cup I_L$.
- A conditional statement `if` $(bexp)$ `goto` l', where $bexp$ is a binary expression (defined below). l' is a program label which indicates the label of the next statement when $bexp$ evaluates to true.
- An assertion statement `assert`$(\bigwedge bexp)$. An assertion holds if $\bigwedge bexp$ evaluates to true. A program is correct if all assertions hold on all executions.

A $bexp$ is either a string or an integer formula defined as follows:

- A string formula can be in two forms: (1) $s \in regexp$, or (2) $s[c_1, c_2] \in regexp$, which specifies that the value of s or the value of the substring (from the c_1^{th} to c_2^{th} character) of s is within a regular language. $s \notin regexp$ is an abbreviation of $s \in regexp'$, where $regexp'$ is the complement set of $regexp$. $s = c$ is an abbreviation of $s \in \{c\}$ and $s \neq c$ is an abbreviation of $s \notin \{c\}$, where c is a constant string.

– An integer formula can be in the form: $\sum_t c_t * i_t \sim c$, where $i_t \in I \cup I_L$ and $\sim \in \{=, <, \leq, \geq, >\}$.

We assume that for each $l : stmt$, $l + 1$ is a valid label if $stmt$ is not a termination statement. For each conditional statement if $(bexp)$ goto l', l' is a valid label.

Modeling the C Example. To analyze normal C programs, one can consider each dereference of a pointer, e.g., $*p$, as a string variable. A sequence value from the address pointed by the pointer is a string value of the string variable. The pointer arithmetic operation, e.g., $p_1 := p_2 + i$, can be considered as a string suffix statement that assigns the suffix of the dereference of p_2 to the dereference of p_1. The previous example can be rewritten using this simple language as follows:

```
strlen(s1){
1: cnt := 0;
2: s2:=s1;
3: if(s2='\0') goto 7;
4: s2:=suffix(s2, 1);
5: cnt := cnt +1;
6: if(s2 != '\0') goto 4;
7: assert(s1.length = cnt);
8: halt;   }
```

3.1   Verification Framework

Assume that $S = \{s_1, \ldots, s_m\}$ and $I = \{i_1, \ldots, i_n\}$ denote the set of string and integer variables in our target program, respectively. In our analysis, each string variable s_k, $1 \leq k \leq m$, is associated with an auxiliary integer variable i_{n+k} as its length $s_k.length$. Hence, we also have the set of auxiliary integer variables $I_L = \{i_{n+1}, \ldots i_{n+m}\}$. A state for each program label consists of a string-automata vector $\boldsymbol{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ and an $n + m$-track arithmetic automaton a.

Each string variable s_k is associated with the string automaton α_k in $\boldsymbol{\alpha}$, which accepts an over approximation of the set of all possible values that s_k can take at the corresponding program label. Each track of the arithmetic automaton a is a binary encoding starting from the least significant bit of the value of an integer variable (the first n tracks) or the value of the length of a string variable (the last m tracks).

A word accepted by the arithmetic automaton corresponds to a valid valuation for the integer variables and the lengths of string variables at the corresponding program point during the execution of the program. The arithmetic automaton accepts an over approximation of the set of possible words at the corresponding program label. Each word w is an assignment of the integer variables and the lengths of the string variables; and each track of w is actually the value that $i \in I \cup I_L$ can take at the corresponding program label. We use $w[k]$ to denote the kth track of the word w. For $1 \leq k \leq n$, $w[k]$ is the value of the integer variable i_k. For $n + 1 \leq k \leq n + m$, $w[k]$ is the length of the string variable s_k. We say a string w is the value of a string variable s_k if $w \in L(\alpha_k)$, and $\exists w' \in L(a)$ such that $w'[k]$ is equal to the binary encoding of $|w|$ starting from the least significant bit.

Forward Fixpoint Computation. Our analysis is based on a standard forward fixpoint computation on $\boldsymbol{\alpha}$ and a for all program labels. For simplicity, we use $\nu[l]$ to denote $\boldsymbol{\alpha}[l]$ and $a[l]$, where $\boldsymbol{\alpha}[l]$ is the string-automaton vector and $a[l]$ is the arithmetic automaton at the program label l. The algorithm is a standard work-queue algorithm as shown in Algorithm 3.

For sequential operations (string/integer assignments), we are continuously computing the post image of $\nu[l]$ against $l : stmt$, and join the result to $\nu[l+1]$ where $l+1$ is the label of the next statement. For branch statement $l : \texttt{if}(bexp)\ \texttt{goto}\ l'$, if the intersection of the language of $\nu[l]$ and $bexp$ is not an empty set, we add the result to $\nu[l']$. If the intersection of the language of $\nu[l]$ and the complement set of $bexp$ is not an empty set, we add the result to $\nu[l+1]$. For checking statement $l : \texttt{assert}(\phi)$, if the language of $\nu[l]$ is not included in ϕ, we raise an alarm.

Upon joining the results, we check whether a fixpoint of that program point is reached. If it is not, we update ν at that program point and push its labeled statement into the queue. Since we target infinite state systems, the fixpoint computation may not terminate. We incorporate an automata widening operator, denoted as ∇_A, proposed by Bartzis and Bultan in [3] to accelerate the fixed point computation. $\nu \nabla \nu'$ is implemented as $\alpha_1 \nabla_A \alpha'_1, \ldots, \alpha_m \nabla_A \alpha'_m$ [16] and $a \nabla_A a'$ [3].

Finally, we detail how to compute post and restrict computations, i.e., post$(\nu, stmt)$ and $\nu \wedge bexp$, in the following paragraphs.

Basic Operations. Before we detail the algorithms of post and restrict computations, we first define some notations and basic operations to simplify our presentation. We use a to denote the arithmetic automaton, and a_k to denote the one-track arithmetic automaton that accepts the values of the kth track of the arithmetic automaton a. We use α to denote a string automaton and $\boldsymbol{\alpha}$ to denote a vector of string automata. α_k is the kth string automaton of $\boldsymbol{\alpha}$. $\texttt{bla}(\alpha)$ returns the binary length automaton of the string automaton α. The binary length automaton can be considered as an one-track arithmetic automaton. We use α^c, where c is an integer constant, to denote the string automaton which accepts arbitrary words having length equal to c. That is $L(\alpha^c) = \{w \mid w \in \Sigma^*, |w| = c\}$. This notation is also extended to a range $[c_1, c_2]$, where c_1, c_2 are integer constants. We say that $\alpha^{[c_1, c_2]}$ is the string automaton that accepts $\{w \mid w \in \Sigma^*, c_1 \leq |w| \leq c_2\}$.

- Extraction: $a \downarrow_k$, returns an one-track arithmetic automaton a_k so that $w \in L(a_k)$ if $\exists w' \in L(a)$ and $w'[k] = w$. a_k is constructed by projecting away all tracks except the kth track of the arithmetic automaton a.
- Projection: $a \uparrow_k$, returns a new arithmetic automaton a' which accepts $\{w \mid w' \in L(a), \forall 1 \leq t \leq m+n, t \neq k, w'[t] = w[t]\}$. a' is constructed by projecting away the track k of the arithmetic automaton a.
- Composition: $a \circ \alpha_k$, returns a new arithmetic automaton a' so that $L(a') = \{w \mid w \in L(a), w[k] \in L(\texttt{bla}(\alpha_k))\}$. a' is constructed by intersecting a with an arithmetic automaton that the track k is accepted by the binary length automaton of the string automaton α_k, and other tracks are unrestricted. This composition restricts $L(a)$ to a smaller set where the length of s_k (the value of the track k) is accepted by the binary length automaton of α_k.

Algorithm 3. COMPOSITEANALYSIS(l_0)

1: Init(ν);
2: queue WQ;
3: WQ.enqueue($l_0 : stmt_0$);
4: **while** $WQ \neq NULL$ **do**
5:    $e := WQ$.dequeue(); Let e be $l : stmt$;
6:    **if** $stmt$ is sequential operation **then**
7:      $tmp := \text{post}(\nu[l], stmt)$;
8:      $tmp := (tmp \cup \nu[l+1])\nabla\nu[l+1]$;
9:      **if** $tmp \not\subseteq \nu[l+1]$ **then**
10:       $\nu[l+1] := tmp$;
11:       WQ.enqueue($l+1$);
12:     **end if**
13:   **end if**
14:   **if** $stmt$ is `if` $bexp$ `goto` l' **then**
15:     **if** CheckIntersection($\nu[l], bexp$) **then**
16:       $tmp := \nu[l] \wedge bexp$;
17:       $tmp := (tmp \cup \nu[l'])\nabla\nu[l']$;
18:       **if** $tmp \not\subseteq \nu[l']$ **then**
19:         $\nu[l'] := tmp$;
20:         WQ.enqueue(l');
21:       **end if**
22:     **end if**
23:     **if** CheckIntersection($\nu[l], \neg bexp$) **then**
24:       $tmp := \nu[l] \wedge \neg bexp$;
25:       $tmp := (tmp \cup \nu[l+1])\nabla\nu[l+1]$;
26:       **if** $tmp \not\subseteq \nu[l+1]$ **then**
27:         $\nu[l+1] := tmp$;
28:         WQ.enqueue($l+1$);
29:       **end if**
30:     **end if**
31:   **end if**
32:   **if** $stmt$ is `assert`(ϕ) **then**
33:     **if** $\neg$ CheckInclusion($\nu[l], \phi$) **then**
34:       Assertion violated!
35:     **end if**
36:   **end if**
37: **end while**

- Boundary: $\min(a_k)$ returns the lower bound of the set of integer values whose binary encodings from the least significant bit are accepted by the one-track automaton a_k. $\max(a_k)$ returns the upper bound.

Post Image. Recall that there are m string variables and n integer variables. Given $stmt$ and the state ν that consists of $\boldsymbol{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ and the arithmetic automaton

a, we want to compute $\alpha' = \langle \alpha'_1, \ldots, \alpha'_m \rangle$ and a' as the result of the post image against $stmt$. We assume that the automata that are not specified remain the same. Let $stmt$ be one of the following:

- $s_k := \mathtt{input}(i_p)$. $\alpha'_k := \alpha^{[c_1, c_2]}$, where $c_1 = \min(a_p)$ and $c_2 = \max(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k} := i_p)$.
- $s_{k_1} := s_{k_2}$. $\alpha'_{k_1} := \alpha_{k_2}$. $a' := \text{CONSTRUCT}(a, i_{n+k_1} := i_{n+k_2})$.
- $s_k := regex$. $\alpha'_k := \text{CONSTRUCT}(regexp)$. $a' := a \uparrow_{n+k} \circ \alpha'_k$.
- $s_{k_1} := \mathtt{prefix}(s_{k_2}, i_p)$. $\alpha'_{k_1} := \text{PREFIX}(\alpha_{k_2}, [c_1, c_2])$, where $c_1 = \min(a_p)$ and $c_2 = \max(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k_1} := i_p) \wedge \text{CONSTRUCT}(i_{n+k_2} - i_p \geq 0)$.
- $s_{k_1} := \mathtt{suffix}(s_{k_2}, i_p)$. $\alpha'_{k_1} := \text{SUFFIX}(\alpha_{k_2}, [c_1, c_2])$, where $c_1 = \min(a_p)$ and $c_2 = \max(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k_1} := i_p) \wedge \text{CONSTRUCT}(i_{n+k_2} - i_p \geq 0)$.
- $s_k := \mathtt{strcat}(s_{k_1}, s_{k_2})$. $\alpha'_k := \text{CONCAT}(\alpha_{k_1}, \alpha_{k_2})$. $a' := \text{CONSTRUCT}(a, i_{n+k} := i_{n+k_1} + i_{n+k_2})$.
- $s_k := \mathtt{replace}(s_{k_1}, s_{k_2}, s_{k_3})$. $\alpha'_k := \text{REPLACE}(\alpha_{k_1}, \alpha_{k_2}, \alpha_{k_3})$. $a' := a \uparrow_{n+k} \wedge a_{tmp}$, where a_{tmp} accepts $\{w \mid w[k] \in L(\mathtt{bla}(\alpha'_k))\}$.
- $i_p := intexp$. $a' := \text{CONSTRUCT}(a, i_p := intexp)$.

Restriction. Here we describe the result of $\nu \wedge bexp$, where ν is the state consists of α and a. Let $bexp$ be one of the following:

- $s_k \in regexp$. $\alpha'_k = \alpha_k \wedge \text{CONSTRUCT}(regexp)$. $a' = a \circ \alpha'_k$.
- $s_k[c_1, c_2] \in regexp$. $\alpha'_k = \alpha_k \wedge \alpha_{tmp}$, where α_{tmp} is constructed by $\text{CONCAT}(\text{CONCAT}(\alpha^{[c_1, c_2]}, \text{CONSTRUCT}(regexp)), \alpha^*)$. $a' = a \circ \alpha'_k$.
- $\sum_t c_t * i_t \sim c$. $\forall t > n. \alpha'_t = \alpha_t \wedge \alpha^{[c_1, c_2]}$, where $c_1 = \min(a' \downarrow_t)$ and $c_2 = \max(a' \downarrow_t)$. $a' = a \wedge \text{CONSTRUCT}(\sum_t c_t * i_t \sim c)$.

3.2 Implementation

Automaton Construction. In this section, we describe how to construct the corresponding arithmetic and string automata used in our composite analysis. The constructions of arithmetic automata including $\text{CONSTRUCT}(\sum_t c_t * i_t \sim c)$ and $\text{CONSTRUCT}(a, i := \sum_t c_t * i_t)$ are detailed in [2]. The latter returns an arithmetic automaton which accepts the result of the post image computation on a against the integer assignment $i := \sum_t c_t * i_t + c$. This construction is implemented by quantifier elimination and variable renaming, i.e., $(\exists i, \Phi(a) \wedge i' = \sum_t c_t * i_t)[I'/I]$. For some special cases, the time complexity of this construction is linear to the size of a [2]. The constructions of string automata including $\text{CONSTRUCT}(regexp)$, $\text{CONCAT}(\alpha_{k_1}, \alpha_{k_2})$, and $\text{REPLACE}(\alpha_{k_1}, \alpha_{k_2}, \alpha_{k_3})$ have been detailed in [16]. We describe the implementation of $\text{PREFIX}(\alpha, [c_1, c_2])$ and the implementation of $\text{SUFFIX}(\alpha, [c_1, c_2])$ below.

Prefix. Formally speaking, α' is a prefix-DFA of α regarding to the range $[c_1, c_2]$, if $L(\alpha') = \{w \mid w \in \Sigma^{[c_1, c_2]}, \exists w', ww' \in L(\alpha)\}$. Given $\alpha = \langle Q, q_0, \Sigma, \delta, F \rangle$ and $[c_1, c_2]$, we first construct $\alpha' = \langle Q, q_0, \Sigma, \delta, F' \rangle$, where $\forall q \in Q, F'(q) =' +'$. α' accepts the prefix of $L(\alpha)$. The next step is restricting its length to the range $[c_1, c_2]$. $\text{PREFIX}(\alpha, [c_1, c_2])$ returns the the result of the intersection of α' and $\alpha^{[c_1, c_2]}$, which is exactly the prefix-DFA of α regarding to the range $[c_1, c_2]$.

Suffix. Formally speaking, α' is a suffix-DFA of α regarding to the range $[c_1, c_2]$, if $L(\alpha') = \{w \mid \exists w' \in \Sigma^{[c_1, c_2]}, w'w \in L(\alpha)\}$. We first introduce the function REACH$(\alpha, [c_1, c_2])$. REACH$(\alpha, [c_1, c_2])$ returns the set of all $[c_1, c_2]$-reachable states. We say a state is $[c_1, c_2]$-reachable if it is reachable from the initial state by k steps and $c_1 \leq k \leq c_2$. Given $\alpha = \langle Q, q_0, \Sigma, \delta, F \rangle$ and $[c_1, c_2]$, we first compute $R =$ REACH$(\alpha, [c_1, c_2])$ via a breadth-first search. We then construct the following finite automaton $\alpha' = \langle Q', q_0', \Sigma, \delta', F' \rangle$, where

- $Q' = Q \cup \{q_0'\}$
- $\forall q, q' \in Q, \delta'(q, \alpha) = q'$, if $\delta(q, \alpha) = q'$.
- $\forall q \in R, q' \in Q, \delta'(q_0', \alpha) = q'$, if $\delta(q, \alpha) = q'$.
- $F'(q_0) =' +'$, if $\exists q \in R, F(q) =' +'$.
- $\forall q \in Q, F'(q) = F(q)$.

Note that α' constructed by the above construction may be a nondeterministic finite automaton. We add auxiliary bits to resolve nondeterminism as proposed in [16]. SUFFIX$(\alpha, [c_1, c_2])$ returns the result of the minimization and determinization of α'.

Boundary. Below we describe how to identify the boundary of a one-track arithmetic automaton, which accepts the binary encodings of a set of integer values from the least significant bit.

Property 4: For an one-track minimized DFA $a = \langle Q, q_0, B^1, \delta, F \rangle$: $\forall q, q' \in Q$, if $\delta(q, 0) = q'$, then $F(q) = F(q')$.

Property 4 states that transitions labelled by 0 cannot change accepting status, which holds due to the fact that by definition, the arithmetic automaton accepts a word and any number of 0 in its higher significant bits. It follows that for any accepted integer value (except 0), the word from the least significant bit up to the most non-zero significant bit of its binary encoding forms a unique path (ended by 1) from the initial state to an accepting state. Furthermore, an accepted non-zero minimal integer value forms the shortest path from the initial state to an accepting state. On the other hand, if there exists an accepted non-zero maximal integer value, the maximal value forms the longest loop-free path from the initial state to an accepting state. Note that if there exists an accepted path containing a loop, a accepts an infinite set and the maximal value does not exist. In this case, we use *inf* to denote the maximal value.

For $\min(a)$ and $\max(a)$, we have implemented two functions MIN(a) and MAX(a). Let m_s be the length of the shortest path that ends with 1 and m_l be the length of the longest loop-free path that ends with 1. Both m_s and m_l can be determined by a breadth first search up to $\sharp Q$ steps. In our implementation, we first check whether a accepts any non-zero integer value. If this is the case, MIN(a) returns $2^{m_s - 1}$, which is a lower bound for the shortest path. If there exists a path containing a loop, MAX(a) returns *inf*. Otherwise MAX(a) returns $2^{m_l + 1} - 1$, which is an upper bound for the longest path. Note that our implementation is a conservative approximation. These bounds can be tightened by tracing the values along paths.

4   Experiments

We experimented with our composite analysis tool on a number of test cases extracted from C string library, buffer overflow benchmarks [10] and web vulnerability benchmarks [16]. These test cases are rather small but involve pointer arithmetic, string content constraints, length constraints, loops, and replacement operations. We manually convert them to our simple imperative language.

For int strlen(char *s), we verify the invariant that the return value is equal to the length of the input string. For char *strrchr(char *s, int c), we verify whether the language accepted by the return string is included in $\{cx \mid x \in \Sigma^*\} \cup \{\epsilon\}$ upon reaching the fixpoint. For buffer overflow benchmarks, we check whether the identified memory may overflow its buffer upon reaching the fixpoint for both buggy (*bad*) and modified (*ok*) cases. For web vulnerability benchmarks, we check whether the identified sensitive function may take any attack string as its input before (*bad*) and after (*ok*) inserting limit constraints and sanitization routines. If it does not, the sensitive function is SQL attack free with respect to the attack pattern Σ^*<scriptΣ^*. Limit constraints are written as new statements that limit the length of string variables using a $limit variable. Table 1 shows that our composite analysis works well in these test cases in terms of both accuracy and performance. As a final remark, for web vulnerability benchmarks, one may restrict limit constraints, e.g., set $limit less than 7, to prevent the specified attacks without adding/modifying sanitization routines. In this case, pure string analysis [16] will raise false alarms.

Table 1. Preliminary experimental results. T: buffer overflow free or SQL attack free.

Test case (*bad*/*ok*)	Result	Time (s)	Memory (kb)
int strlen(char *s)	T	0.037	522
char *strrchr(char *s, int c)	T	0.011	360
gxine (CVE-2007-0406)	F/T	0.014/0.018	216/252
samba (CVE-2007-0453)	F/T	0.015/0.021	218/252
MyEasyMarket-4.1 (trans.php:218)	F/T	0.032/0.041	704/712
PBLguestbook-1.32 (pblguestbook.php:1210)	F/T	0.021/0.022	496/662
BloggIT 1.0 (admin.php:27)	F/T	0.719/0.721	5857/7067

5   Conclusion

We presented an automata-based approach for symbolic verification of infinite-state systems with unbounded string and integer variables. Our approach combines string and size analyses and is able to verify properties that cannot be verified with either analysis alone. We demonstrated the effectiveness of our approach on several examples.

References

1. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kruegel, C., Kirda, E., Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: Proceedings of the Symposium on Security and Privacy (2008)
2. Bartzis, C., Bultan, T.: Efficient symbolic representations for arithmetic constraints in verification. Int. J. Found. Comput. Sci. 14(4), 605–624 (2003)
3. 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)
4. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
5. Dor, N., Rodeh, M., Sagiv, M.: Cssv: towards a realistic tool for statically detecting all buffer overflows in c. SIGPLAN Not. 38(5), 155–167 (2003)
6. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting sql injection vulnerabilities. In: COMPSAC 2007: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Washington, DC, USA, vol. 1, pp. 87–96. IEEE Computer Society Press, Los Alamitos (2007)
7. Ganapathy, V., Jha, S., Chandler, D., Melski, D., Vitek, D.: Buffer overrun detection using linear programming and static analysis. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 345–354 (2003)
8. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: 35th ACM Symposium on Principles of Programming Languages, pp. 235–246. ACM, New York (2008)
9. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, Tucson, AZ, USA, pp. 339–348 (2008)
10. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, pp. 389–392 (2007)
11. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: TAICPART-MUTATION 2007: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, Washington, DC, USA, pp. 13–22. IEEE Computer Society Press, Los Alamitos (2007)
12. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Network and Distributed System Security Symposium, pp. 3–17 (2000)
13. Wolper, P., Boigelot, B.: On the construction of automata from linear arithmetic constraints. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 1–19. Springer, Heidelberg (2000)
14. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX-SS 2006: Proceedings of the 15th conference on USENIX Security Symposium, Berkeley, CA, USA, pp. 13–13. USENIX Association (2006)
15. Xu, R.-G., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: ISSTA 2008: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM Press, New York (2008)
16. Yu, F., Bultan, T., Cova, M., Ibarra, O.H.: Symbolic string verification: An automata-based approach. In: 15th International SPIN Workshop on Model Checking of Software (2008)

Iterating Octagons

Marius Bozga, Codruţa Gîrlea, and Radu Iosif

VERIMAG/CNRS, 2 Avenue de Vignate, 38610 Gières, France
{bozga,girlea,iosif}@imag.fr

Abstract. In this paper we prove that the transitive closure of a non-deterministic octagonal relation using integer counters can be expressed in Presburger arithmetic. The direct consequence of this fact is that the reachability problem is decidable for flat counter automata with octagonal transition relations. This result improves the previous results of Comon and Jurski [7] and Bozga, Iosif and Lakhnech [6] concerning the computation of transitive closures for difference bound relations. The importance of this result is justified by the wide use of octagons to computing sound abstractions of real-life systems [15]. We have implemented the octagonal transitive closure algorithm in a prototype system for the analysis of counter automata, called FLATA, and we have experimented with a number of test cases.

1   Introduction

Counter automata (register machines) are widely investigated models of computation. Since the result of Minsky [16] showing Turing-completeness of 2-counter machines, research on counter automata pursued in two directions. The first one is defining subclasses of counter automata for which various decision problems (e.g. reachability, emptiness, boundedness, disjointness, containment, equivalence) are found to be decidable. Examples include Reversal-bounded Counter Machines [13], Petri Nets and Vector Addition Systems [18] or Flat Counter Automata [14]. Often, decidability of various problems is achieved by defining the set of reachable configurations in a decidable logic, such as Presburger arithmetic [17]. Such definitions are precise, i.e. no information is lost by the use of over-approximation.

Another, orthogonal, direction of work is concerned with finding sound (but not necessarily complete) answers to the decision problems mentioned above, in a cost-effective way. Such approaches use abstract domains (Polyhedra [8], Octagons [15], Difference Constraints [1], etc.) and compute fixed points that are over-approximations of the set of reachable configurations.

Both approaches to the analysis of counter automata benefit, in some sense, from algorithms for computing transitive closures of arithmetic relations. If I is the initial set of configurations, and R is the transition relation of the counter automaton, then $R^*(I)$ (the image of I through R^*) is the set of reachable configurations, where $R^* = \bigcup_{n \geq 0} R^n$ is the transitive closure of R. The problem lies essentially in expressing the infinite disjunction from the definition of R^*, in a finite way.

In this paper we consider octagonal transition relations that are conjunctions of atoms of the form $\pm x \pm y \leq c$, where x and y are counter values, either at the current step, or at the next step (in which case they are denoted by primed variables), and $c \in \mathbb{Z}$ is an integer constant. For this class of relations, we prove that the transitive closure is expressible in Presburger arithmetic [17]. This improves the previous result of Comon and Jurski [7], showing that the transitive closure of a difference bound constraint (a conjunction of atoms of the form $x - y \leq c$, with x, y possible primed counters, and $c \in \mathbb{Z}$) is Presburger-definable.

We adopt the classical representation of octagonal constraints (or octagons) $\varphi(x_1, \ldots, x_n)$ as difference bound constraints $\varphi(y_1, \ldots, y_{2n})$, where y_{2i-1} stands for $+x_i$ and y_{2i} for $-x_i$, and the implicit condition $y_{2i-1} + y_{2i} = 0$, $1 \leq i \leq n$. With this convention, [2] provides an algorithm for computing the canonical form of an octagon, by first computing the strongest closure of the corresponding difference bound constraint (using the classical Floyd-Warshall cubic algorithm), and subsequently tightening the constraints of the form $y_{2i} - y_{2i-1} \leq c$ and $y_{2i-1} - y_{2i} \leq d$ by adjusting c to $2\lfloor c/2 \rfloor$ and d to $2\lfloor d/2 \rfloor$, respectively[1]. We apply this idea to tightening octagonal relations of the form R^k, where R is an octagon and $k \geq 1$ is the arbitrary number of iterations, obtaining in this way a Presburger formula $\psi(k, x_1, ..., x_n, x'_1, ..., x'_n)$ equivalent to the k-th iteration of φ. The transitive closure of R is thus $R^* = \exists k.\psi$.

The main application of this result is that the following problems are decidable, for the class of counter automata with octagonal transition rules:

- *reachability*: the automaton has a run leading from a configuration in I to a configuration in S, where I and S are Presburger-definable sets of configurations.
- *emptiness*: the automaton has at least a run starting with all counters set to zero, and leading to a final control location.

In particular, decidability of the reachability problem is useful for the verification of safety properties (e.g. assertion checking) of integer programs, whereas the emptiness problem is a promising approach for the analysis of programs with integer arrays. Indeed, the works of [11,5] reduce the satisfiability problem for two logics on integer arrays to the emptiness problem of a counter automaton. By enlarging the class of counter automata for which this problem is decidable, we enhance the expressiveness of (decidable) logics of integer arrays.

Finally, we have implemented our method in a tool for the analysis of counter automata, called FLATA [10]. In particular, FLATA computes the transitive closure of elementary octagonal cycles, which is used in computing the set of reachable configurations for flat counter automata. We have experimented with a number of test cases and provided several experimental results.

---

[1] This is needed because we consider the counters to range over integer numbers. If rationals or real number would have been used, this would not have been needed.

1.1   Related Work

The domain of counter machines has been investigated starting with the seminal work of Minsky [16]. His result on Turing-completeness of 2-counter machines motivated research on subclasses of counter automata, for which some problems are decidable, such as: Reversal-bounded Counter Machines [13], Petri Nets and Vector Addition Systems [18] or Flat Counter Automata [14].

The class of Flat Counter Automata which is closest to our work is the one studied by Comon and Jurski in [7]. In this work, they prove that the transitive closure of a difference bound constraint is Presburger-definable. In [6] we showing how to effectively compute the transitive closure of a difference bound constraint directly as a finite set of linear inequation systems, opening thus the possibility of using SMT tools for the analysis of such models.

Octagonal constraints are investigated in the comprehensive paper of Miné [15]. Among other results, his paper presents a cubic time tightening algorithm for an octagonal constraint, which is an improvement of the classical algorithm of Harvey and Stuckey [12]. However, this algorithm is not suitable to tightening octagonal constraints of parametric size, as the ones we obtain by iteration. For this reason, we adapt the (newer) tightening algorithm of Bagnara, Hill and Zaffanella [2] to octagons obtained by arbitrary iteration, as they prove that it is sufficient to adjust the constants *after* the computation of minimal paths (by the Floyd-Warshall algorithm).

Our paper extends to octagons the class of non-deterministic relations for which the transitive closure can be effectively expressed in Presburger arithmetic. This result relies essentially on our method from [6] to computing minimal paths in constraint graphs obtained by iteration, and the idea of [2] for tightening (finite) octagonal relations.

On a different line of work, Boigelot [3], and Finkel and Leroux [9] have studied the computation of transitive closure for affine relations of the form $\mathbf{x}' = A\mathbf{x} + \mathbf{b}$. Their class of systems differs in nature from ours, in what their transition relations are functional (deterministic), whereas octagons are not. Because of this, their results seem incomparable to ours.

Roadmap. The paper is organized as follows. Section 2 introduces difference bound constraints and difference bound relations, recalling a number of results on DBMs, Section 3 defines octagonal constraints, and gives a necessary quantifier elimination result on octagons, while Section 4 is dedicated to our main result, namely computing the transitive closure of an octagonal relation. Section 5 gives implementation results, and finally, Section 6 concludes. Due to space reasons, all proofs are given in [4].

2   Difference Bounds

In this section we recall several definitions and results on difference bound constraints, and the transitive closure of difference bound relations. In the rest of the paper, $\mathbb{Z}$ denotes the set of integer numbers, $n > 0$ is the number of variables

and $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is the tuple of variables. If $\varphi(\mathbf{x})$ is a logical formula in which $x_1, x_2, \ldots, x_n$ occur free, and $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is a tuple of values, $\varphi[\mathbf{v}/\mathbf{x}]$ denotes the formula in which each occurrence of x_i has been replaced by v_i. We denote by $\mathbf{v} \models \varphi$ the fact that $\varphi[\mathbf{v}/\mathbf{x}]$ is logically equivalent to true. We say that φ is *consistent* if there exists at least one $\mathbf{v} \in \mathbb{Z}^n$ such that $\mathbf{v} \models \varphi$, and *inconsistent* otherwise. If φ is a formula, let $AP(\varphi)$ denote the set of atomic propositions in φ. If m is a matrix, then $m_{i,j}$ denote the element of m situated at line i and column j.

2.1 Difference Bound Constraints

Definition 1. *Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be a set of variables ranging over* $\mathbb{Z}$. *Then a formula* $\phi(\mathbf{x})$ *is a* difference bound constraint *if it is equivalent to a finite conjunction of atomic propositions of the form* $x_i - x_j \leq \alpha_{i,j}$, $i \neq j$, $1 \leq i, j \leq n$, *where* $\alpha_{i,j} \in \mathbb{Z}$.

For instance, $x - y = 5$ is a difference bound constraint, as it is equivalent to $x - y \leq 5 \wedge y - x \leq -5$. In practice difference bound constraints are represented either as matrices or as graphs, each of these representations being suitable for particular procedures (e.g. closure, iteration). We define these representations and procedures below.

Definition 2. *Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be a set of variables ranging over* $\mathbb{Z}$ *and* $\phi(\mathbf{x})$ *be a difference bound constraint. Then a* difference bound matrix *(DBM) representing* ϕ *is an* $n \times n$ *matrix* m *such that:*

$$m_{i,j} = \begin{cases} \alpha_{i,j} & \text{if } (x_i - x_j \leq \alpha_{i,j}) \in AP(\phi) \\ \infty & \text{otherwise} \end{cases}$$

For a $n \times n$ DBM m we denote by $\gamma(m) = \{\mathbf{v} \in \mathbb{Z}^n \mid v_i - v_j \leq m_{i,j}, 1 \leq i, j \leq n\}$ the set of *concretizations* of m. Notice that this is exactly the set of models of the corresponding difference bound constraint φ, i.e. $\gamma(m) = \{\mathbf{v} \mid \mathbf{v} \models \varphi\}$. A DBM m is said to be consistent if $\gamma(m) \neq \emptyset$, and inconsistent otherwise.

Definition 3. *Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be a set of variables ranging over* $\mathbb{Z}$ *and* $\phi(\mathbf{x})$ *be a difference bound constraint. Then* ϕ *can be represented as a weighted graph* $\mathcal{G}$ *with vertices* $x_1, x_2, \ldots, x_n$ *in which there is an arc with weight* $\alpha_{i,j}$ *between* x_i *and* x_j *in* $\mathcal{G}$ *if there is a constraint* $x_i - x_j \leq \alpha_{i,j}$ *in* ϕ. *This graph is also called a* constraint graph.

Whenever the graph $\mathcal{G}$ is obvious from the context, we will denote by $x \xrightarrow{\alpha} y$ the fact that there exists an edge with weight α from x to y in $\mathcal{G}$. Notice that the DBM m of a difference bound constraint is the incidence matrix of its corresponding constraint graph. This graph will be denoted as $\mathcal{G}(m)$ in the following. The three notions (constraint, matrix, graph) are related by the following property:

Property 1. Let ϕ be a difference bound constraint, m the corresponding DBM and $\mathcal{G}(m)$ the corresponding graph. Then the following three statements are equivalent :

1. ϕ is inconsistent
2. $\mathcal{G}(m)$ contains at least one negative weight cycle
3. $\gamma(m) = \emptyset$

On one hand, the DBM representation of a difference bound constraint is suitable for computing its normal form, i.e. the most "explicit" formula that has the same set of models.

Definition 4. *An $n \times n$ consistent DBM m is said to be* closed *if and only if the following hold:*

1. $m_{i,i} = 0 \ \forall 1 \leq i \leq n$
2. $m_{i,j} \leq m_{i,k} + m_{k,j} \ \forall 1 \leq i, j, k \leq n$

The *shortest path closure* of a consistent DBM m is a closed DBM m^* such that $\gamma(m) = \gamma(m^*)$. It is well-known that, if m is consistent then m^* is unique, and it can be computed from m in time $\mathcal{O}(n^3)$, by the classical Floyd-Warshall algorithm. Moreover, the following condition holds, for all $1 \leq i, j \leq n$:

$$m_{i,j}^* = \min \left\{ \sum_{l=0}^{k-1} m_{i_l, i_{l+1}} | x_i = x_{i_0} x_{i_1} ... x_{i_k} = x_j \text{ is a path in } \mathcal{G}(m) \right\}$$

On the other hand, the graph representation of a difference bound constraint is suitable for existential quantifier elimination[2]. Concretely, given a difference bound constraint $\phi(\mathbf{x})$, the formula $\exists x_k.\phi$ is a difference bound constraint as well, and its corresponding graph is effectively computable from the graph of ϕ, as shown by the following property :

Property 2. Let $\mathbf{x} = (x_1, x_2, ..., x_n)$ be a set of variables ranging over $\mathbb{Z}$, $\phi(\mathbf{x})$ be a difference bound constraint and m be the corresponding $n \times n$ DBM. Then, for any $1 \leq k \leq n$ the formula $\exists x_k.\phi(\mathbf{x})$ is a difference bound constraint, and its corresponding constraint graph is obtained by erasing the vertex x_k together with the incident arcs from the graph $\mathcal{G}(m^*)$. Moreover, the DBM of the resulting graph is also closed.

The importance of this result will be made clear in the next section, because it directly implies that the class of difference bound relations is closed under composition. This is the first ingredient of our transitive closure method.

---

[2] In general, difference bound constraints are not closed under universal quantification, however finite disjunctions of difference bound constraints are, since the negation of a difference bound constraint is always equivalent to a finite disjunction of difference constraints.

2.2   Difference Bound Relations

Definition 5. *Let* $\mathbf{x} = (x_1, x_2, ..., x_n)$, $\mathbf{x}' = (x_1', x_2', ..., x_n')$ *be sets of variables ranging over* $\mathbb{Z}$. *A relation* $R(\mathbf{x}, \mathbf{x}')$ *is a* difference bound relation *if it is equivalent to a finite conjunction of terms of the form* $x_i - x_j \leq a_{i,j}, x_i' - x_j \leq b_{i,j}$, $x_i - x_j' \leq c_{i,j}$, *or* $x_i' - x_j' \leq d_{i,j}$, *where* $a_{i,j}, b_{i,j}, c_{i,j}, d_{i,j} \in \mathbb{Z}$, $1 \leq i, j \leq n$.

According to the previous section, a difference bound relation can be represented as a constraint graph with nodes in the set $\mathbf{x} \cup \mathbf{x}'$, such that $x \xrightarrow{\alpha} y$ if and only if $x - y \leq \alpha \in AP(R)$, for all $x, y \in \mathbf{x} \cup \mathbf{x}'$. Just like before, the incidence matrix of this graph is the DBM of the relation.

In the following, we denote $\mathbf{x}^{(l)} = (x_1^{(l)}, x_2^{(l)}, \ldots, x_n^{(l)})$, $\mathbf{x}^{(\geq l)} = \bigcup_{s \geq l} \mathbf{x}^{(s)}$ and $\mathbf{x}^{(\leq l)} = \bigcup_{s \leq l} \mathbf{x}^{(s)}$, for any $l \geq 0$. Given a difference bound relation $R(\mathbf{x}, \mathbf{x}')$, we define the k-th iteration of R:

$$R^k(\mathbf{x}^{(0)}, \mathbf{x}^{(k)}) = \exists \mathbf{x}^{(1)} \ldots \exists \mathbf{x}^{(k-1)}.R(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}) \wedge \ldots \wedge R(\mathbf{x}^{(k-1)}, \mathbf{x}^{(k)})$$

for all $k \geq 1$. The transitive closure of R is defined as $R^* = \exists k.R^k$.

Notice that, since existential quantifiers can be eliminated from difference bound constraints (cf. Theorem 2), difference bound relations are closed under composition, and therefore R^k is a difference bound relation, for any $k \geq 0$.

We now recall a result from [6], namely that the k-th iteration R^k of a difference bound relation R is equivalent to a finite set of linear inequation systems. As a result, R^* is directly definable in Presburger arithmetic.

The constraint graph $\mathcal{G}^k$, representing the matrix of R^k (i.e. $R(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}) \wedge \ldots \wedge R(\mathbf{x}^{(k-1)}, \mathbf{x}^{(k)})$) is the graph composed of k connected copies of the constraint graph of R.

Definition 6. *Let* $\mathbf{x} = (x_1, x_2, ..., x_n)$, $\mathbf{x}' = (x_1', x_2', ..., x_n')$ *be sets of variables ranging over* $\mathbb{Z}$ *and* $R(\mathbf{x}, \mathbf{x}')$ *a DBM relation. The constraint graph of* R^k *has as set of vertices* $\bigcup\limits_{l=0}^{k} \mathbf{x}^{(l)}$ *and, for all* $1 \leq i, j \leq n$, *for all* $0 \leq l < k$:

$$x_i^{(l)} \xrightarrow{a} x_j^{(l)} \iff x_i - x_j \leq a \in AP(R)$$

$$x_i^{(l)} \xrightarrow{a} x_j^{(l+1)} \iff x_i - x_j' \leq a \in AP(R)$$

$$x_i^{(l+1)} \xrightarrow{a} x_j^{(l)} \iff x_i' - x_j \leq a \in AP(R)$$

$$x_i^{(l+1)} \xrightarrow{a} x_j^{(l+1)} \iff x_i' - x_j' \leq a \in AP(R)$$

In order to compute R^k, for a given $k \geq 1$, we must first compute the closure of the constraint graph from definition 6, in order to remove the intermediary nodes $\mathbf{x}^{(l)}$, $0 < l < k$. According to Theorem 2, R^k is a difference bound constraint, and the graph obtained after eliminating the intermediary variables is the constraint graph of R^k. Therefore, R^k can be written as a conjunction of the form:

$$\bigwedge_{1\leq i,j\leq n} x_i^{(0)} - x_j^{(0)} \leq \min_k\{x_i^{(0)} \rightsquigarrow x_j^{(0)}\} \wedge \bigwedge_{1\leq i,j\leq n} x_i^{(0)} - x_j^{(k)} \leq \min_k\{x_i^{(0)} \rightsquigarrow x_j^{(k)}\} \wedge$$
$$\bigwedge_{1\leq i,j\leq n} x_i^{(k)} - x_j^{(k)} \leq \min_k\{x_i^{(k)} \rightsquigarrow x_j^{(k)}\} \wedge \bigwedge_{1\leq i,j\leq n} x_i^{(k)} - x_j^{(0)} \leq \min_k\{x_i^{(k)} \rightsquigarrow x_j^{(0)}\}$$

where $\min_k\{x \rightsquigarrow y\} \in \mathbb{Z}\cup\{\pm\infty\}$ denotes the value of the minimal path between nodes x and y in the constraint graph of R^{k}[3]. Notice that R^k is satisfiable if and only there are no cycles of negative cost within the constraint graph from Definition 6 (cf. property 1). Since this graph is obtained by connecting k copies of R, if there is a negative cycle in the graph, there will also be a negative cycle that goes through $x_i^{(0)}$, for some $1 \leq i \leq n$.

The main result of [7,6] is that $\min_k\{x_i^{(0)} \rightsquigarrow x_j^{(0)}\}$, $\min_k\{x_i^{(0)} \rightsquigarrow x_j^{(k)}\}$, $\min_k\{x_i^{(k)} \rightsquigarrow x_j^{(k)}\}$ and $\min_k\{x_i^{(k)} \rightsquigarrow x_j^{(0)}\}$ are Presburger definable functions.
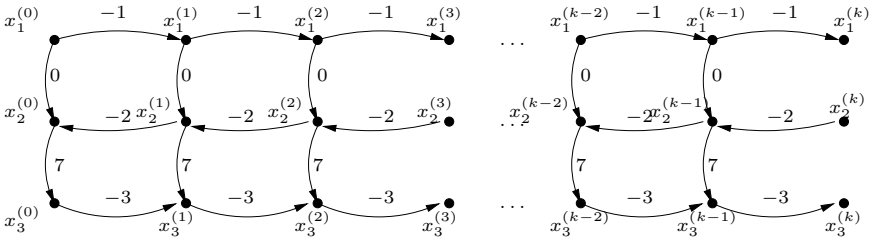
Fig. 1. Graph representation of R^k, $k = 1, 2, 3, \ldots$

Example 1. Let $R(\mathbf{x}, \mathbf{x}')$ be the following DBM relation, where $\mathbf{x} = \{x_1, x_2, x_3\}$:

$$R(\mathbf{x}, \mathbf{x}') = x_1 - x_1' \leq -1 \wedge x_1 - x_2 \leq 0 \wedge x_2 - x_3 \leq 7 \wedge x_2' - x_2 \leq -2 \wedge x_3 - x_3' \leq -3$$

Figure 1 shows the graph representation of R^k, for $k = 1, 2, 3\ldots$. By computing the minimal weight paths in the constraint graph of R^k for $k = 1, 2, 3, \ldots$ we obtain, e.g. :

$$R^{(1)}(\mathbf{x}, \mathbf{x}') = x_1 - x_2 \leq 0 \wedge x_1 - x_3 \leq 7 \wedge x_2 - x_3 \leq 7 \wedge x_1 - x_1' \leq -1 \wedge x_1 - x_3' \leq 4 \wedge$$
$$x_2 - x_3' \leq 4 \wedge x_3 - x_3' \leq -3 \wedge x_2' - x_2 \leq -2 \wedge x_2' - x_3 \leq 5 \wedge x_2' - x_3' \leq 2$$
$$R^{(2)}(\mathbf{x}, \mathbf{x}') = x_1 - x_2 \leq -3 \wedge x_1 - x_3 \leq 4 \wedge x_2 - x_3 \leq 7 \wedge x_1 - x_1' \leq -2 \wedge x_1 - x_3' \leq -2 \wedge$$
$$x_2 - x_3' \leq 1 \wedge x_3 - x_3' \leq -6 \wedge x_2' - x_2 \leq -4 \wedge x_2' - x_3 \leq 3 \wedge x_2' - x_3' \leq -3$$
$$R^{(3)}(\mathbf{x}, \mathbf{x}') = x_1 - x_2 \leq -6 \wedge x_1 - x_3 \leq 1 \wedge x_2 - x_3 \leq 7 \wedge x_1 - x_1' \leq -3 \wedge x_1 - x_3' \leq -8 \wedge$$
$$x_2 - x_3' \leq -2 \wedge x_3 - x_3' \leq -9 \wedge x_2' - x_2 \leq -6 \wedge x_2' - x_3 \leq 1 \wedge x_2' - x_3' \leq -8$$
$$R^{(n\geq 4)}(\mathbf{x}, \mathbf{x}') = x_1 - x_2 \leq -3 - 3(n-2) \wedge x_1 - x_3 \leq -2 - 3(n-4) \wedge x_2 - x_3 \leq 7 \wedge$$
$$x_1 - x_1' \leq -n \wedge x_1 - x_3' \leq -14 - 6(n-4) \wedge x_2 - x_3' \leq 1 - 3(n-2) \wedge$$
$$x_3 - x_3' \leq -3n \wedge x_2' - x_2 \leq -2n \wedge x_2' - x_3 \leq 3 - 2(n-2) \wedge$$
$$x_2' - x_3' \leq -3 - 5(n-2) \qquad \qquad \square$$

---

[3] If $\min_k\{x \rightsquigarrow y\} = -\infty$ the constraint is logically equivalent to false, and if $\min\{x \rightsquigarrow y\} = \infty$ it is logically equivalent to true.

3   Octagonal Constraints

This section is dedicated to octagonal constraints. We provide preliminary results that are needed to define transitive closures of octagonal relations, in the next section.

Definition 7. *Let* $\mathbf{x} = (x_1, x_2, ..., x_n)$ *be a set of variables ranging over* $\mathbb{Z}$. *Then a formula* $\phi(\mathbf{x})$ *is an* octagonal constraint *if it is equivalent to a finite conjunction of terms of the form* $\pm x_i \pm x_j \leq \alpha_{i,j}$, $2x_i \leq \beta_i$, *or* $-2x_i \leq \delta_i$, *where* $\alpha_{i,j}, \beta_i, \delta_i \in \mathbb{Z}$ *and* $i \neq j, 1 \leq i, j \leq n$.

The name *octagon* comes from the fact that, in two dimensions, these constraints can be graphically represented by polyhedra with at most eight edges. We represent octagons using the set of variables $\mathbf{y} = (y_1, y_2, \ldots, y_{2n})$, with the convention that y_{2i-1} stands for x_i and y_{2i} for $-x_i$, respectively. For instance, the octagonal constraint $x_1 + x_2 = 3$ is represented as $y_1 - y_4 \leq 3 \wedge y_2 - y_3 \leq -3$.

If we denote by $\overline{\phi} = \phi[y_1/x_1, y_2/-x_1, \ldots, y_{2n-1}/x_n, y_{2n}/-x_n]$, we obtain the following valid entailment: $\phi(\mathbf{x}) \rightarrow (\exists y_2, y_4, \ldots, y_{2n}.\overline{\phi})[x_1/y_1, \ldots, x_n/y_{2n-1}]$. Moreover $\phi(\mathbf{x}) \leftrightarrow (\exists y_2, y_4, \ldots, y_{2n}.\overline{\phi} \wedge \bigwedge_{i=1}^{n} y_{2i-1} + y_{2i} = 0)[x_1/y_1, \ldots, x_n/y_{2n-1}]$.

To handle the $\mathbf{y}$ variables in the following, we define $\overline{i} = i - 1$, if i is even, and $\overline{i} = i + 1$ if i is odd. Obviously, we have $\overline{\overline{i}} = i$, for all $i \in \mathbb{Z}$, $i \geq 0$. The following definition extends the matrix representation from difference bound constraints to octagons:

Definition 8. *Let* $\mathbf{x} = (x_1, x_2, ..., x_n)$ *be a set of variables ranging over* $\mathbb{Z}$ *and* $\phi(\mathbf{x})$ *be an octagonal constraint. Then an* octagonal difference bound matrix *or* octagonal DBM *representing* ϕ *is an* $2n \times 2n$ *matrix* m *such that:*

$$
\begin{aligned}
(x_i - x_j \leq \alpha_{i,j}) \in AP(\phi) &\iff m_{2i-1,2j-1} = m_{2j,2i} = \alpha_{i,j} \\
(-x_i - x_j \leq \alpha_{i,j}) \in AP(\phi) &\iff m_{2i,2j-1} = m_{2j,2i-1} = \alpha_{i,j} \\
(-x_i + x_j \leq \alpha_{i,j}) \in AP(\phi) &\iff m_{2i,2j} = m_{2j-1,2i-1} = \alpha_{i,j} \\
(x_i + x_j \leq \alpha_{i,j}) \in AP(\phi) &\iff m_{2i-1,2j} = m_{2j-1,2i} = \alpha_{i,j} \\
(2x_i \leq \beta_i) \in AP(\phi) &\iff m_{2i-1,2i} = \beta_i \\
(-2x_i \leq \delta_i) \in AP(\phi) &\iff m_{2i,2i-1} = \delta_i
\end{aligned}
$$

A $2n \times 2n$ octagonal DBM m is said to be *coherent* if and only if $m_{i,j} = m_{\overline{j},\overline{i}}$, for all $1 \leq i, j \leq 2n$. This property is needed since any constraint $x_i - x_j \leq \alpha$, $1 \leq i, j \leq n$ can be represented as both $y_{2i-1} - y_{2j-1} \leq \alpha$ and $y_{2j} - y_{2i} \leq \alpha$. If m is coherent, we denote by

$$\gamma^{Oct}(m) = \{(v_1, v_2, ..., v_n) \in \mathbb{Z}^n \mid (v_1, -v_1, v_2, -v_2, ..., v_n, -v_n) \in \gamma(m)\}$$

the set of concretizations, i.e. the set of models of the octagonal constraint represented by m. Also, m is said to be consistent if $\gamma^{Oct}(m) \neq \emptyset$, and inconsistent otherwise. The *octagonal graph* of an octagonal constraint $\phi(\mathbf{x})$, represented as a difference bound constraint $\overline{\phi}(\mathbf{y})$ with DBM m, has vertices $\mathbf{y}$, and an arc labeled by $m_{i,j}$ between y_i and y_j if and only if $m_{i,j} < \infty$, $1 \leq i, j \leq 2n$.

Definition 9. *A consistent coherent* $2n \times 2n$ *DBM m in* $\mathbb{Z}$ *is said to be* tightly closed *if and only if the following hold:*

1. $m_{i,i} = 0 \ \forall 1 \le i \le 2n$
2. $m_{i,\bar{i}}$ *is even* $\forall 1 \le i \le 2n$
3. $m_{i,j} \le m_{i,k} + m_{k,j} \ \forall 1 \le i, j, k \le 2n$
4. $m_{i,j} \le (m_{i,\bar{i}} + m_{\bar{j},j})/2 \ \forall 1 \le i, j \le 2n$

The intuition behind the last point is the following: since $y_i - y_{\bar{i}} \le m_{i,\bar{i}}$ and $y_{\bar{j}} - y_j \le m_{\bar{j},j}$ and $y_i = -y_{\bar{i}}$, $y_j = -y_{\bar{j}}$ if we interpret the DBM m as an octagonal constraint, we have that $y_i \le \lfloor \frac{m_{i,\bar{i}}}{2} \rfloor$ and $-y_j \le \lfloor \frac{m_{\bar{j},j}}{2} \rfloor$, hence $y_i - y_j \le \lfloor \frac{m_{i,\bar{i}}}{2} \rfloor + \lfloor \frac{m_{\bar{j},j}}{2} \rfloor$. Therefore, the tightening of an octagonal DBM has to come up with values for $m_{i,j}$ that are smaller than $\lfloor \frac{m_{i,\bar{i}}}{2} \rfloor + \lfloor \frac{m_{\bar{j},j}}{2} \rfloor$, for all $1 \le i, j \le 2n$.

The tight closure of a $2n \times 2n$ octagonal DBM m is an octagonal DBM m_t^* such that $\gamma^{Oct}(m) = \gamma^{Oct}(m_t^*)$, and m_t^* is tightly closed. By Theorem 7 in [15], we know that the tight closure of a consistent octagonal DBM exists and is unique.

We now characterize the consistency of octagonal constraints using their DBM representations. The following result, proved in [2], is crucial for the developments of the next section, therefore we cite (a slightly modified version of) it here:

Theorem 1 ([2]). *Let m be a* $2n \times 2n$ *octagonal DBM consistent and coherent, and let* m^* *be its closure. Suppose that* $\left\lfloor \frac{m_{i,\bar{i}}^*}{2} \right\rfloor + \left\lfloor \frac{m_{\bar{i},i}^*}{2} \right\rfloor \ge 0$, *for all* $1 \le i \le 2n$. *Then* $\gamma^{Oct}(m) \neq \emptyset$ *and the DBM* m^T *defined as:*

$$m_{i,j}^T = min \left\{ m_{i,j}^*, \left\lfloor \frac{m_{i,\bar{i}}^*}{2} \right\rfloor + \left\lfloor \frac{m_{\bar{j},j}^*}{2} \right\rfloor \right\}$$

for all $1 \le i, j \le 2n$, *is the tight closure of m.*

It follows immediately that an octagonal constraint is consistent if and only if (1) its constraint graph representation does not contain negative weight cycles and moreover, (2) if the sum between the halved weights of the minimal paths from any node x_i to $x_{\bar{i}}$, and from $x_{\bar{i}}$ and x_i is positive. Notice that the lack of negative weight cycles alone is not enough, as shown by the following example.

Example 2. Let $\phi(x, y, z, x', y', z') = (x - y' \le 1) \wedge (y' + x \le -2) \wedge (-x + z' \le 1) \wedge (-z' - x \le 0)$ be an octagonal constraint on integer variables x, y, z and x', y', z'. The constraint is inconsistent since it implies $(-1 \le 2x \le -1)$ and thus has no integer solution. However, its corresponding octagonal DBM does not contain negative weight cycles. The tightening step, i.e. replacing each element $m_{i\bar{i}}$ with $2 \lfloor \frac{m_{i\bar{i}}}{2} \rfloor$, exhibits two negative weight cycles, between $+x$ and $-x$, and between $+z'$ and $-z'$. $\qquad\square$

We are now ready to give the result of existential quantification over variables occurring within octagons. Namely, we prove that, if $\phi(x_1, \dots, x_n)$ is an octagonal constraint, the formula $\exists x_k.\phi$, for some $1 \le k \le n$ is again an octagonal

constraint, and its representation is effectively computable from the constraint graph of ϕ.

In the following, we assume w.l.o.g. that $k = n$ (if this is not the case, we proceed to reindexing variables). From now on assume that $\phi(x_1, \ldots, x_n)$ is consistent, and let m be its $2n \times 2n$ tightly closed coherent octagonal DBM. We denote by $\beta(m)$ the $2n - 2 \times 2n - 2$ matrix from which the $2n - 1$ and $2n$ lines and columns have been eliminated. Notice that these correspond to the x_n and $-x_n$ terms, respectively.

Lemma 1. *Let $m \in \mathbb{Z}_{2n \times 2n}$ be the coherent and tightly closed difference bound matrix for $\phi(x_1, \ldots, x_n)$. Then $\beta(m)$ is also coherent and tightly closed.*

The following theorem proves that octagons are closed under existential quantification.

Theorem 2. *Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a set of variables ranging over $\mathbb{Z}$, $\phi(\mathbf{x})$ a consistent octagonal constraint and m the corresponding $2n \times 2n$ octagonal DBM. Then formula $\exists x_k . \phi(\mathbf{x})$ is equivalent to erasing vertices y_{2k} and y_{2k-1} together with the incident arcs from the graph $\mathcal{G}(m_t^*)$. Moreover, the resulting graph is also coherent and tightly closed.*

4   Octagonal Relations

This section is dedicated to our main result, the definition of the transitive closure of an octagonal relation in Presburger arithmetic. An octagonal relation is defined in a similar way to a difference bound relation.

Definition 10. *Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{x}' = (x_1', x_2', \ldots, x_n')$ be sets of variables ranging over $\mathbb{Z}$. Then an octagonal relation $R(\mathbf{x}, \mathbf{x}')$ is a relation that can be written as a finite conjunction of terms of the form $\pm x_i \pm x_j \leq a_{i,j}, \pm x_i' \pm x_j \leq b_{i,j}, \pm x_i \pm x_j' \leq c_{i,j}, \pm x_i' \pm x_j' \leq d_{i,j}, \pm 2x_i \leq e_{i,j}$ or $\pm 2x_i' \leq f_{i,j}$, where $a_{i,j}, b_{i,j}, c_{i,j}, d_{i,j}, e_{i,j}, f_{i,j} \in \mathbb{Z}$ and $1 \leq i, j \leq n$, $i \neq j$.*

The DBM and graph representation of an octagonal relation are defined as in the previous case of difference bound relations. For the rest of this section, let $R(\mathbf{x}, \mathbf{x}')$ be an octagonal relation and $\overline{R}(\mathbf{y}, \mathbf{y}')$ be its difference bound representation. Let $\mathcal{G}$ be the graph of $\overline{R}$, $\overline{R}^k$ be the k-th iteration of $\overline{R}$, m_k be its DBM, and $\mathcal{G}^k$ be the graph corresponding to $\overline{R}^k$, obtained by connecting k copies of $\mathcal{G}$. The formal definition of $\mathcal{G}^k$ is similar to Definition 6.

In order to compute R^k, we need two ingredients. First, we need to check for consistency of the unfolded relation, that is we need to check that $\gamma^{Oct}(m_k) \neq \emptyset$. Second, we need to obtain the strongest octagonal constraints between $\mathbf{y}^{(0)}$ and $\mathbf{y}^{(k)}$ in $\mathcal{G}^k$. By Theorem 2, all intermediate vertices $\mathbf{y}^{(l)}$, $0 < l < k$ can be eliminated from $\mathcal{G}^k$, and the result is a tightly closed octagonal graph, whose interpretation is equivalent to R^k.

By Theorem 1, both points require the computation of the values $(m_k^*)_{i^{(l)}, \overline{i^{(l)}}}$ and $(m_k^*)_{\overline{i^{(l)}}, i^{(l)}}$, for all $1 \leq i \leq 2n$ and $0 \leq l \leq k$, where the index $i^{(l)}$ refers to
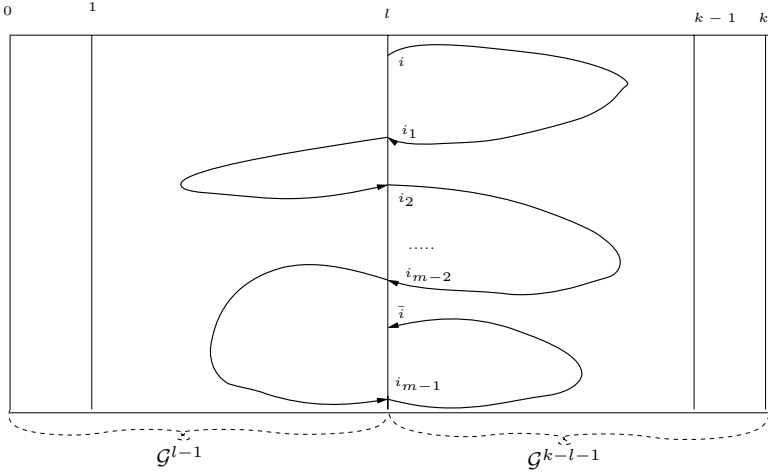
Fig. 2. Computing paths of weight $(m_k{}^*)_{i^{(l)}, \overline{i^{(l)}}}$

the variable $y_i^{(l)}$ in the DBM representation of $\overline{R}(\mathbf{y}, \mathbf{y}')$. But these values can now be defined by Presburger formulae, using the results from [7,6]. To understand this point, consider the situation depicted in Figure 2.

Assume in the following that $\mathcal{G}^k$ has no cycles of negative weight. The absence of negative cycles can be checked a-priori using e.g. the method in [6]. Since we are aiming at computing minimal weight paths, it is sufficient to consider acyclic paths only[4]. For a fixed $0 < l < k$, an acyclic path between the nodes $y_i^{(l)}$ and $y_{\bar{i}}^{(l)}$, for some $1 \le i \le 2n$, can be decomposed in at most $2n-1$ segments starting and ending in $\mathbf{y}^{(l)}$, but not intersecting with $\mathbf{y}^{(l)}$, other than in the beginning and in the end. Moreover, if the path considered is of minimal weight, these segments are of minimal weight as well.

Let $1 \le i = i_0, i_1, \ldots, i_m = \bar{i} \le 2n$, be a set of pairwise distinct indices, such that

$$(m_k^*)_{i^{(l)}, \overline{i^{(l)}}} = \sum_{j=0}^{m-1} (m_k^*)_{i_j^{(l)}, i_{j+1}^{(l)}}$$

As in Figure 2, several of the paths $y_i^{(l)} \rightsquigarrow y_{i_1}^{(l)}, y_{i_1}^{(l)} \rightsquigarrow y_{i_2}^{(l)}, \ldots, y_{i_{m-1}}^{(l)} \rightsquigarrow y_{\bar{i}}^{(l)}$ will contain only nodes from the set $\mathbf{y}^{(\le l)}$, whereas the rest will contain only nodes from the set $\mathbf{y}^{(\ge l)}$. Notice that, the paths from $\mathbf{y}^{(\le l)}$ connect the terminal nodes of $\mathcal{G}^{l-1}$, whereas the paths from $\mathbf{y}^{(\ge l)}$ connect the initial nodes of $\mathcal{G}^{k-l-1}$. Therefore, we have:

- $(m_k^*)_{i_j^{(l)}, i_{j+1}^{(l)}} = \min_{l-1}\{y_{i_j}^{(l)} \rightsquigarrow y_{i_{j+1}}^{(l)}\}$, if $y_{i_j}^{(l)} \rightsquigarrow y_{i_{j+1}}^{(l)}$ belongs to $\mathbf{y}^{(\le l)}$, and
- $(m_k^*)_{i_j^{(l)}, i_{j+1}^{(l)}} = \min_{k-l-1}\{y_{i_j}^{(0)} \rightsquigarrow y_{i_{j+1}}^{(0)}\}$, if $y_{i_j}^{(l)} \rightsquigarrow y_{i_{j+1}}^{(l)}$ belongs to $\mathbf{y}^{(\ge l)}$.

---

[4] If a path has a cycle of a positive weight, it cannot be minimal.

But since the $\min_j\{x \leadsto y\}$ functions are definable in Presburger arithmetic [7,6], we obtain that $(m_k^*)_{i,\bar{i}}$ are Presburger definable as well. Since moreover, integer division can be defined in Presburger as

$$\left\lfloor \frac{u}{2} \right\rfloor = v \iff 2v \le u \le 2v + 1$$

it is possible to encode the consistency check of Theorem 1 by a Presburger formula, and effectively perform the check required by Theorem 1:

$$\bigwedge_{\substack{1 \le i \le 2n \\ 0 \le l \le k}} \left\lfloor \frac{m^*_{i^{(l)},\overline{i^{(l)}}}}{2} \right\rfloor + \left\lfloor \frac{m^*_{\overline{i^{(l)}},i^{(l)}}}{2} \right\rfloor \ge 0$$

The formula used to check consistency of $R(\mathbf{x}, \mathbf{x}')^k$, $k \ge 0$ is of size $\mathcal{O}(2^{2n \log 2n})$, where n is the number of variables in $\mathbf{x}$. For more details, the interested reader is pointed to [4].

After quantifier elimination, the strongest octagonal relations are obtained by taking the restriction of the tight closure of m_k to $\mathbf{y}^{(0)}$ and $\mathbf{y}^{(k)}$, according to Theorem 1 and Theorem 2:

$$y_i^{(0)} - y_j^{(k)} \le (m_{kt}^*)_{i^{(0)},j^{(k)}} = min\left\{ (m_{kt}^*)_{i^{(0)},j^{(k)}}, \left\lfloor \frac{(m_{kt}^*)_{i^{(0)},\overline{i^{(0)}}}}{2} \right\rfloor + \left\lfloor \frac{(m_{kt}^*)_{\overline{j^{(k)}},j^{(k)}}}{2} \right\rfloor \right\}$$

$$y_i^{(k)} - y_j^{(0)} \le (m_{kt}^*)_{i^{(k)},j^{(0)}} = min\left\{ (m_{kt}^*)_{i^{(k)},j^{(0)}}, \left\lfloor \frac{(m_{kt}^*)_{i^{(k)},\overline{i^{(k)}}}}{2} \right\rfloor + \left\lfloor \frac{(m_{kt}^*)_{\overline{j^{(0)}},j^{(0)}}}{2} \right\rfloor \right\}$$

$$y_i^{(0)} - y_j^{(0)} \le (m_{kt}^*)_{i^{(0)},j^{(0)}} = min\left\{ (m_{kt}^*)_{i^{(0)},j^{(0)}}, \left\lfloor \frac{(m_{kt}^*)_{i^{(0)},\overline{i^{(0)}}}}{2} \right\rfloor + \left\lfloor \frac{(m_{kt}^*)_{\overline{j^{(0)}},j^{(0)}}}{2} \right\rfloor \right\}$$

$$y_i^{(k)} - y_j^{(k)} \le (m_{kt}^*)_{i^{(k)},j^{(k)}} = min\left\{ (m_{kt}^*)_{i^{(k)},j^{(k)}}, \left\lfloor \frac{(m_{kt}^*)_{i^{(k)},\overline{i^{(k)}}}}{2} \right\rfloor + \left\lfloor \frac{(m_{kt}^*)_{\overline{j^{(k)}},j^{(k)}}}{2} \right\rfloor \right\}$$

Since we showed that all weights of the form $m^*_{i^{(l)},\overline{i^{(l)}}}$ are definable in Presburger arithmetic, it follows that R^k is also Presburger definable. Consequently, the transitive closure of R is the Presburger formula $\exists k.R^k$, leading to the following theorem:

Theorem 3. *The transitive closure of an integer octagonal relation is Presburger definable.*

Example 3. Let $R(\mathbf{x}, \mathbf{x}')$ be the following octagonal relation, where $\mathbf{x} = \{x_1, x_2\}$

$$R(\mathbf{x}, \mathbf{x}') = x_1 + x_2 \le 5 \wedge x_1' - x_1 \le -2 \wedge x_2' - x_2 \le -3$$

Figure 3 shows the graph representation of R^k for $k = 1, 2, 3, \ldots$. The n-step closure is:

$$R^n(\mathbf{x}, \mathbf{x}') = x_1 + x_2 \le 5 \wedge x_1' - x_1 \le -2n \wedge x_2' - x_2 \le -3n \wedge$$
$$x_1 + x_2' \le 5 - 3n \wedge x_2 + x_1' \le 5 - 2n \wedge x_1' + x_2' \le 5 - 5n \qquad \square$$
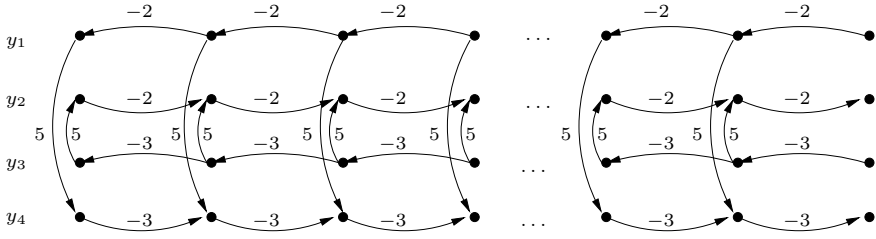
Fig. 3. Graph representation of $\overline{R}^k$, $k = 1, 2, 3, \ldots$

5 Implementation and Experience

We have implemented the method for computing transitive closures of octagonal relations in a tool for the analysis of counter automata, that we are currently

Table 1. Experimental results

R	R^n	*time (ms)*	
		$-check$	$+check$
$x_1 - x_2 \leq 0$ $x_2 - x_3 \leq 7$ $x_1 - x_1' \leq -1$ $x_3 - x_3' \leq -3$ $x_2' - x_2 \leq -2$	$x_1 - x_2 \leq 3 - 3n$ $x_1 - x_3 \leq 10 - 3n$ $x_2 - x_3 \leq 7$ $x_1 - x_1' \leq -n$ $x_1 - x_3' \leq 10 - 6n$ $x_2 - x_3' \leq 7 - 3n$ $x_3 - x_3' \leq -3n$ $x_2' - x_2 \leq -2n$ $x_2' - x_3 \leq 7 - 2n$ $x_2' - x_3' \leq 7 - 5n$	383	
$x_1 + x_2' \leq -1$ $-x_2 - x_1' \leq -2$	$n \equiv_2 0 \Rightarrow x_1 - x_1' \leq -3n/2$ $n \equiv_2 0 \Rightarrow x_2' - x_2 \leq -3n/2$ $n \equiv_2 1 \Rightarrow x_1 + x_2' \leq -1 - 3(n-1)/2$ $n \equiv_2 1 \Rightarrow -x_2 - x_1' \leq -2 - 3(n-1)/2$	75	116
$x_1 - x_1' \leq 0$ $x_1 + x_2' \leq -1$ $-x_2 - x_1' \leq -2$	$n \equiv_2 0 \Rightarrow x_1 - x_1' \leq -3n/2$ $n \equiv_2 1 \Rightarrow x_1 - x_1' \leq -3(n-1)/2$ $n \equiv_2 0 \Rightarrow x_2' - x_2 \leq -3n/2$ $n \equiv_2 1 \wedge n \geq 3 \Rightarrow x_2' - x_2 \leq -3 - 3(n-3)/2$ $n \equiv_2 0 \wedge n \geq 2 \Rightarrow x_1 + x_2' \leq -1 - 3(n-2)/2$ $n \equiv_2 1 \Rightarrow x_1 + x_2' \leq -1 - 3(n-1)/2$ $n \equiv_2 0 \wedge n \geq 2 \Rightarrow -x_2 - x_1' \leq -2 - 3(n-2)/2$ $n \equiv_2 1 \Rightarrow -x_2 - x_1' \leq -2 - 3(n-1)/2$	81	124
$x_1 + x_2 \leq 5$ $x_1' - x_1 \leq -2$ $x_2' - x_2 \leq -3$	$x_1 + x_2 \leq 5$ $x_1' - x_1 \leq -2n$ $x_2' - x_2 \leq -3n$ $x_1 + x_2' \leq 5 - 3n$ $x_2 + x_1' \leq 5 - 2n$ $x_1' + x_2' \leq 5 - 5n$	11686	38729

developing. Table 1 shows several octagonal relations (first column), the closed form of their iteration (second column), the execution times (in milliseconds) needed to compute the closed forms without consistency checks (third column), and with consistency checks (fourth column). Notice that the first relation is the difference bound relation example from Section 2, for which no consistency check is needed.

The tool, called FLATA, is implemented in Java$^{(TM)}$, and it is currently available at [10]. The execution times from Table 1 are relative to a Intel$^{(R)}$ Xeon$^{(TM)}$ CPU 3.00GHz equipped with 1 Gb of RAM. As an approximate upper bound, we can in principle handle difference bound relations on ten counters, and/or octagons on five counters, within reasonable execution times ($\approx$ 10 min).

The fourth example from Table 1 takes significantly more time than the previous ones. This happens because in the pre-processing step we need to replace $x_1 + x_2 \leq 5$ with $x_1 - x_3' \leq 5 \wedge x_3' + x_2 \leq 0$, where x_3 is a fresh variable, thus having to deal with a DBM relation with 6 counters.

6   Conclusions

We have considered the problem of computing transitive closures of octagonal relations, which are finite conjunctions of atomic formulae of the form $\pm x \pm y \leq c$, where x and y are possibly primed integer variables. We show that the k-th iteration of such a relation has a Presburger definable closed form. As a consequence, the transitive closure is also Presburger definable. This result enlarges the class of non-deterministic counter automata for which the reachability problem is decidable, to counter automata with octagonal transition relations. This result is expected to have an impact in the fields of software verification, as abstract models of software systems are described using octagons. We have implemented our method in a tool for the analysis of counter automata, and report on a number of experiments.

References

1. Bagnara, R.: Data-Flow Analysis for Constraint Logic-Based Languages. Ph. D. Thesis, Dipartimento di Informatica, Università di Pisa (1997)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: An improved tight closure algorithm for integer octagonal constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 8–21. Springer, Heidelberg (2008)
3. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. TCS 309(2), 413–468 (2003)
4. Bozga, M., Girlea, C., Iosif, R.: Iterating octagons. TR VERIMAG (2008)
5. Bozga, M., Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: LPAR 2008. LNCS(LNAI), vol. 5330, pp. 558–573. Springer, Heidelberg (2008)
6. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 577–588. Springer, Heidelberg (2006)

7. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–97. ACM Press, New York (1978)
9. Finkel, A., Leroux, J.: How to compose presburger-accelerations: Applications to broadcast protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
10. http://www-verimag.imag.fr/~async/flata/flata.html
11. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 474–489. Springer, Heidelberg (2008)
12. Harvey, W., Stuckey, P.: A unit two variable per inequality integer constraint solver for constraint logic programming. In: Australian Computer Science Conference, pp. 102–111 (1997)
13. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. Journal of the ACM 25(1), 116–133 (1978)
14. Leroux, J., Sutre, G.: Flat counter automata almost everywhere? In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 489–503. Springer, Heidelberg (2005)
15. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
16. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967)
17. Presburger, M.: Über die Vollstandigkeit eines gewissen Systems der Arithmetik. In: Comptes rendus du I Congrés des Pays Slaves, Warsaw (1929)
18. Reutenauer, C.: Aspects Mathématiques des Réseaux de Petri. Collection Études et Recherches en Informatique. Masson (1989)

Verifying Reference Counting Implementations⋆

Michael Emmi[1], Ranjit Jhala[2], Eddie Kohler[1], and Rupak Majumdar[1]

[1] University of California, Los Angeles
{mje,kohler,rupak}@cs.ucla.edu
[2] University of California, San Diego
jhala@cs.ucsd.edu

Abstract. Reference counting is a widely-used resource management idiom which maintains a count of references to each resource by incrementing the count upon an acquisition, and decrementing upon a release; resources whose counts fall to zero may be recycled. We present an algorithm to verify the correctness of reference counting with minimal user interaction. Our algorithm performs compositional verification through the combination of symbolic *temporal case splitting* and predicate abstraction-based reachability. Temporal case splitting reduces the verification of an unbounded number of processes and resources to verification of a finite number through the use of *Skolem variables*. The finite state instances are discharged by symbolic model checking, with an auxiliary invariant correlating reference counts with the number of held references. We have implemented our algorithm in Referee, a reference counting analysis tool for C programs, and applied Referee to two real programs: the memory allocator of an OS kernel and the file interface of the Yaffs file system. In both cases our algorithm proves correct the use of reference counts in less than one minute.

1 Introduction

Reference counting is a widely-used resource management idiom where the references to each resource unit (*e.g.,* memory cell, file handle, device structure) are counted. The programmer increments the count when acquiring a resource and decrements it when releasing. A resource may be recycled when its reference count reaches zero.

Despite its ubiquity, reference counting is difficult to implement correctly. Ensuring a resource is not accessed after its count reaches zero requires precisely reasoning about shared heap objects in concurrent programs with a statically unknown number of sharers. In the most benign case, errors in reference counting cause resource leaks: when the last reference to an object is removed but the reference count is not decremented to zero. More dangerous errors can allow unprivileged read or write access to critical regions of memory that have been inappropriately reclaimed and recycled, possibly compromising sensitive information.

We show how predicate-abstraction based software model checking can be extended with compositional reasoning techniques to enable the static verification of the correctness of reference counting implementations (*i.e.,* that accessed objects have positive counts). The problem is difficult as such programs are unbounded in several dimensions;

first, an unbounded number of objects may be dynamically allocated, second, each unit may be accessed concurrently by an unbounded number of sharers, and hence, third, the reference count for each individual object may grow without bound. These complications prohibit the direct application of finite-state techniques such as model checking, to verify reference counting. Furthermore, standard program analysis abstractions that summarize an unbounded number of dynamic objects (*e.g.,* clients, resources) are too imprecise since they do not *count* the objects they summarize.

Our approach for verifying reference counting implementations follows the following strategy. As a first step, we perform *compositional reasoning* to reduce the verification problem to a number of finite-state verification problems whose combined validity implies the original program's [22]. One possible verification strategy is to tag each resource with a handle, and ensure that clients only access resources to which they have handles. Correctness then follows separately from the correctness of handling each resource, and each handling process. This first step is called *temporal case splitting*: we check validity for a particular tracked resource and a particular accessee in an environment that abstracts all other resources and accessors.

Temporal case splitting trades the original complex verification problem for an infinite number of separate simpler verification obligations. However, using symmetry, we observe that discharging the proof obligation for an *arbitrary* symbolically identified resource, and an *arbitrary* symbolically identified client, implies discharging each of the infinitely many obligations induced by case splitting. Thus, as a second step, we use *Skolem variables* to name single, but arbitrary, resources and clients. The Skolem variables induce a natural finite abstraction of the system which distinguishes only the fixed resources bound to Skolem variables (and abstracts all other resources). Similarly, instead of tracking every client, the abstraction tracks only the fixed clients bound to Skolem variables, abstracting the effects of other clients. Skolemization enables *strong updates* on the tracked resources: we can follow each increment or decrement to the tracked resource precisely; all updates to the other resources are *weak*: their effects are unknown.

Unfortunately, the strategies given so far are still insufficient; we must also deal with unbounded reference count *values*. On the one hand, abstracting a counter's increments and decrements by untracked clients results in a loss of the precise counter value, and a proof is generally not possible. On the other hand, the abstract domain remains infinite (with a different value for each counter value) if we track each of these writes precisely. To solve this problem we observe that correctness follows from knowing a unit's count is positive if and only if it is referenced by some client. To prove this, as a third step, we introduce a *reference predicate*, specifying the *meaning* of referencing a unit, and automatically insert an *auxiliary variable* whose value, by construction (i.e. instrumentation) equals the number of client mappings satisfying the reference predicate. An *auxiliary invariant* enforces a positive valued auxiliary counter whenever a client satisfies the reference predicate.

With these steps, we reduce proving an object's real reference count positive to checking that (1) the tracked client satisfies the reference predicate, and (2) the real reference count *equals* the auxiliary count. As (1) follows by precisely tracking the truth value of the reference predicate for the tracked client, and (2) follows from reasoning

about the equality of program variables, the resulting obligation can be discharged using well-established techniques: either through specially designed *abstract domains* [6] that can prove linear relationships among variables, or, as we implement, through predicate abstraction and counterexample-guided refinement [11,5,1,13] based model checking. The meta-argument used for the auxiliary invariant is manually proved sound outside of the program analysis, and can be *reused* to verify any program that implements reference counting, once the reference predicate is specified. The meta-argument used for the auxiliary invariant is manually proved sound outside of the program analysis, and can be instantiated with different reference predicates to verify any program that implements reference counting. Moreover, the soundness of our approach is independent of the choice of reference predicate: an invalid predicate yields a failed proof, since either (1) or (2) will fail to hold.

In summary, our analysis combines four ingredients:

Temporal case splitting to reduce the (infinite state) verification goal over infinitely many objects to infinitely many (finite state) subgoals over individual objects,

Skolemization to reduce infinitely many verification subgoals over different objects, into a single verification goal over an *arbitrary* object,

Auxiliary state to provide a finite representation of unbounded execution history *i.e.,* the unbounded reference count for a given object, and

Model checking to discharge the finite state verification goals induced by the use of temporal case splitting, skolemization and auxiliary state.

While the general techniques have been known [22], we have not seen them successfully applied in software model checking so far, and believe that our implementation is an interesting application of compositional verification to a relevant systems problem.

We have implemented these ideas in a static analyzer for verifying sound reference counting in C programs. Given a program, and a user-specified set of Skolem variables, our tool instruments the program with auxiliary state and auxiliary invariant instantiation, and performs model checking on the instrumented program abstracted by Skolem- and predicate-abstraction. The model checking engine of our tool is based on the software model checker BLAST [13], and uses an iterative refinement of the abstract transition relation based on counterexample traces [12,15]. Our technique is not completely automatic, and requires that the user identify reference counted datatypes as well as which variables to perform case-splits on. In our experiments, we have found the identification of reference counted datatypes and case-split variables can be performed with a limited knowledge of the program. Our analysis relieves the programmer of the difficult burden of providing precise inductive assertions at function and loop boundaries, a task which is readily performed by the model checker.

We have applied our tool to two case studies: the virtual memory subsystem of the JOS operating system kernel [17], and the file handle interface routines of the YAFFS file system [23]. In each case, the soundness argument depends on precise reasoning about arbitrarily many clients acquiring and releasing resources. These modules (each of a few hundred lines) encapsulate reference counting; sound counting within them implies sound counting for the entire systems. Each example can be verified by our tool within a minute.

2   Verification Technique

We now formalize our verification technique and illustrate with an example.

Preliminaries: Programs and Safety. For our formal presentation, we assume an abstract representation of programs by transition systems [21]. A *program* $P = \langle X, L, \ell_0, R \rangle$ consists of a set X of variables, a set L of control locations, an initial location $\ell_0 \in L$, and a transition relation R. Variables in X have values over integers or functions. (Functions are used to model (unbounded) arrays by mapping natural numbers, *i.e.,* the "indices", to values.) A transition $\langle \ell, \rho, \ell' \rangle \in R$ is a move from control location ℓ to location ℓ', satisfying ρ, a constraint over free variables from $X \cup X'$. The variables from X denote values at location ℓ, and the variables from X' denote the values of variables from X at location ℓ'. The sets of locations and transitions naturally define a directed graph, called the program's *control-flow graph* (CFG).

A *data state* of the program P is a valuation of the variables from X; the set of all data states is denoted Σ. We use constraints to represent sets of data states. For a constraint ρ over $X \cup X'$ and a valuation $\langle s, s' \rangle \in \Sigma \times \Sigma'$, we write $\langle s, s' \rangle \vDash \rho$ if the valuation satisfies the constraint ρ. A *state* $\langle \ell, s \rangle$ consists of a location $\ell \in L$ and a data state s. A *computation* of the program P is a sequence of states $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots, \langle \ell_k, s_k \rangle \in (L \times \Sigma)^*$, where ℓ_0 is the initial location and for each $i \in \{0, \ldots, k-1\}$, there is a transition $\langle \ell_i, \rho, \ell_{i+1} \rangle \in R$ such that $\langle s_i, s_{i+1} \rangle \vDash \rho$. A data state s is *reachable* at location ℓ if $\langle \ell, s \rangle$ appears in some computation. A state $\langle \ell, s \rangle$ is reachable if the data state s is reachable at location ℓ. Let φ be a set of states. A program P is *safe* w.r.t. φ iff all reachable states of P are contained in φ.

Example. Figure 1 shows an abstraction of a shared memory system in which an arbitrary number of processes (syntactically identified with pid) share an unbounded number of resources, indexed by g, and reference counted by the array count. For readability, we present programs in a C-like syntax instead of as tuples. All reference counts are initially zero. Each process first chooses a resource (line 1), then acquires the resource while incrementing its reference count (line 2), performs some task, then releases the resource while decrementing its reference count (line 5). We assume lines 2 and 5 execute atomically. Although implicit, the system may "recycle" a resource when its reference count reaches 0; to ensure the system does not recycle live resources, we seek to verify the validity of the assertion on line 3. The simple reference count example is "obviously correct". However, consider a modified version where the acquire of the resource and the increment of the reference count are not performed atomically, but in distinct steps. This implementation is buggy: between the resource acquisition and the reference count increment, the resource can be freed, if another process happens to hold the only other reference to the same resource, and calls **decref**. It follows that the **incref** operation can read and write on freed (or worse, reallocated) memory. Similar bugs have been found in Windows device drivers [24].

For this simple example the assertion always holds because the current process at line 3 holds a reference to resource g, and hence count[g]≥1. The assertion is an instance of a safety property, and can be checked by ensuring that no reachable program state violates it. Unfortunately, there are infinitely many reachable states of the system

Initially

```
count[g] = 0 for all g
```

Process pid

```
1   choose g;
2   ref[pid] ← g; incref count[g];
3   assert(count[g] > 0);
4   do work
5   ref[pid] ← −1; decref count[g];
```

Fig. 1. Abstract reference counting

```
1    atomic {
2      item ← acquire(0);
3      incref(item);
4    }
5    repeat {
6      choose g;
7      atomic {
8        new_item ← acquire(g);
9        decref(item);
10       item ← new_item;
11       incref(item);
12     }
13   }
```

Fig. 2. Buggy reference counting

Initially

```
count[g] = 0 for all g
x_Π[g] = 0 for all g
```

Process pid

```
1   choose g;
2   ref[pid] ← g; incref count[g];
        update_aux(x_Π[g], ref[pid]=g)
3   assert(pid=P ⇒ g=G ⇒ count[g]>0);
4   do work; update_aux(x_Π[g], ref[pid]=g)
5   ref[pid] ← −1; decref count[g];
        update_aux(x_Π[g], ref[pid]=g)
```

Fig. 3. Abstract reference counting, after Skolemization and Auxiliary Instrumentation. The programmer manually identifies the Skolem variables P and G. The system automatically inserts the auxiliary variables and instrumentation.

as the set of resources, processes, and counter values are all unbounded. Hence we must perform reachability analysis over an *abstraction* of the system.

Here the usual abstraction techniques for arrays [1,13], such as merging all elements into a single element, are too imprecise; they prohibit the analysis from performing strong updates (*i.e.,* precisely tracking information about a resource), and from distinguishing individual resources. Similarly, to prove the assertion we would require an abstract domain that could distinguish the infinitely many states where count[g] has different values. For example, a predicate abstraction [11] based domain would have to track an unbounded number of predicates of the form count[g]=n for each index g and each integer value n that can be stored in count[g].

Step 1: Temporal Case Splitting

Temporal case splitting [22] is a proof technique that decomposes the proof of a program property into sub-proofs, one for each value in the domain of a particular variable. It is based on the following observation.

Lemma 1. (Case Splitting) *Let x be a variable of program P, and φ a set of states. Then P is safe w.r.t. φ iff for each c in the domain of x, the program P is safe w.r.t. $(x = c) \Rightarrow \varphi$.*

Temporal case splitting can be *nested*: in order to check safety w.r.t. $(x = i) \Rightarrow \varphi$, we can further case split on a second variable, and so on.

Example. For the example of Figure 1, we may split the assertion on line 3 into an infinite number of assertions **assert** (g=0 $\Rightarrow$ count[g]>0), **assert** (g=1 $\Rightarrow$ count[g]>0), and so on, one for each resource. By the same reasoning, we can case split further over the process identifier into an infinite number of assertions **assert** (pid=0 $\Rightarrow$ g=0 $\Rightarrow$ count[g]>0),   **assert** (pid=1 $\Rightarrow$ g=0 $\Rightarrow$ count[g]>0),   and so on, one for each process and resource pair. Temporal case splitting is sound in that if each subgoal is true, then the original safety property is also true. However, by itself it is not very useful, as it introduces an infinite number of sub-goals.

Step 2: Skolemization

Though case splitting introduces infinitely many sub-goals, the sub-goals are *symmetric* as each process behaves in a manner similar to the others, and the resources are distinct copies of the same entity. Instead of checking each concrete process and resource separately, we can perform a *single* check for an *arbitrary* process and an *arbitrary* resource. If we prove this goal, then the assertion is valid *for all* processes and all resources. To *name* the arbitrary (but fixed) process and resource, require that the programmer identify *Skolem variables*. These are fresh variables, distinct from the original program variables, that are non-deterministically initialized with an arbitrary value from a possibly unbounded range, and not modified subsequently.

Formally, we introduce Skolem variables as follows. Let $P = \langle X, L, \ell_0, R \rangle$ be a program, and let S be a set of *Skolem variables* disjoint from X. We denote by $P[S] = \langle X \cup S, L, \ell_0, R[S] \rangle$ the program P augmented with Skolem variables S, where $\langle \ell, \rho', \ell' \rangle \in R[S]$ iff there is a transition $\langle \ell, \rho, \ell' \rangle \in R$ and $\rho' \equiv \rho \wedge \bigwedge_{s \in S} s' = s$. An extended data state is a valuation to $X \cup S$, an extended state consists of a location ℓ and an extended data state. To distinguish states of P from states of $P[S]$ (which additionally contain valuations to the Skolem variables), we qualify states with the programs by writing P-state, or $P[S]$-state. By definition, the Skolem variables do not alter the program's behavior; they exist solely for the purpose of the proof and need not be maintained at runtime.

Lemma 2. (Skolemization) *Let x be a variable of program P, φ a set of P-states, S a set of Skolem variables, and $s \in S$. P is safe w.r.t. φ iff $P[S]$ is safe w.r.t. $(x = s) \Rightarrow \varphi$.*

Proof. First of all, P is safe w.r.t φ iff $P[S]$ is. By Lemma 1 $P[S]$ is safe w.r.t. φ iff $P[S]$ is safe w.r.t. $(x = c) \Rightarrow \varphi$ for each c in the domain of x. Since the set of states $(x = c) \Rightarrow \varphi$ is equal to $(x = s \wedge s = c) \Rightarrow \varphi$ (s is not assigned to in $P[S]$), and thus equal to $(s = c) \Rightarrow (x = s) \Rightarrow \varphi$, $P[S]$ is safe w.r.t. φ iff $P[S]$ is safe w.r.t. $(s = c) \Rightarrow (x = s) \Rightarrow \varphi$ for each c in the domain of x, and again by Lemma 1, iff $P[S]$ is safe w.r.t. $(x = s) \Rightarrow \varphi$.

Example. For the program of Figure 1, we (manually) identify two Skolem variables corresponding to the unbounded arrays of processes and resources: P corresponds to an arbitrary process, and G corresponds to an arbitrary resource. Since G and P are never assigned to (they do not even exist in the original program), the infinite number of assertions **assert** (pid=i $\Rightarrow$ g=j $\Rightarrow$ count[g] > 0), one for each i and j, are equivalent to

the single assertion **assert**(pid=P $\Rightarrow$ g=G $\Rightarrow$ count[g]>0), because G=0 $\vee$ G=1 $\vee \cdots$, and P=0 $\vee$ P=1 $\vee \cdots$ are both valid formulæ.

The key benefit of the Skolem variables is that they induce a sound finite abstraction on the state space. Instead of a possibly unbounded number of processes, we (strongly) track the *single* process whose identifier is equal to P, and effectively merge all the other processes (whose identifiers are different from P) into one abstract "summary" process. Similarly, instead of an unbounded number of indices of the count array, we strongly track the resource at index G, and merge the cells whose index is different from G into a single summary cell. For example, using predicate abstraction, we would track the predicate ref[P]=G, rather than ref[p1]=G, ref[p2]=G, ..., effectively dividing these process-specific facts into the fact at P, and those in any other untracked process.

Example. Consider the following C program:

```
1: for (i = 0; i < N; i++) a[i] = 0;
2: for (i = 0; i < N; i++) assert(a[i] == 0);
```

To verify the assert on Line 2, the analysis must infer that the loop on Line 1 initializes *all* the cells with indices between 0 and N-1 with the value 0. Instead of reasoning about an unbounded number of cells, suppose the programmer introduces a skolem variable S, that represents an arbitrary index into the array. Case splitting w.r.t. S replaces the assertion on Line 2 with **assert**(i==S => a[i]==0). That is, the verification is reduced to an assertion over the single array cell S and all others are ignored. Finally, notice that predicate abstraction over predicates 0≤i, i<N, 0≤S, S<N, S<i, S=i, S>i, and a[S]=0 suffices to prove the reduced assertion. Using these predicates, the analysis infers that at Line 1, the invariant (a): (0≤S ∧ S<N ∧ S<i) $\Rightarrow$ a[S]=0 holds, using which it it infers that at Line 2, the invariant (b): (0≤S ∧ S<N) $\Rightarrow$ a[S]=0 holds. Finally, it infers that at the assert, (0≤i ∧ i<N), which with (b) proves the assert. By the choice of predicates, we made the analysis precisely track the cell indexed by S, while merging (*i.e.,* ignoring) the values of all other cells.

The choice of Skolems affects the precision but not the soundness of our technique. A poor choice can yield an abstraction that is too coarse for verification. A simple heuristic is to choose a Skolem for each unbounded object (e.g. processes, array indices).

Step 3: Auxiliary Variables and Invariants
We need one more step before applying model checking: strengthening the program transition relation using auxiliary invariants.

Formally, let $P = \langle X, L, \ell_0, R \rangle$ be a program, S a set of Skolem variables for P, Y be a set of *auxiliary variables* disjoint from $X \cup S$, and for each $y \in Y$, an *auxiliary update function* ϕ_y mapping current and next values of $X \cup S$ and current values of Y to a value in the domain of y. A *monitored program* $P[S, Y, \phi] = \langle X \cup S \cup Y, L, \ell_0, R[S, Y] \rangle$ has a transition relation $R[S, Y]$ such that $\langle \ell, \rho, \ell' \rangle \in R[S]$ iff $\langle \ell, \rho', \ell' \rangle \in R[S, Y]$ where $\rho' \Leftrightarrow \rho \wedge \bigwedge_{y \in Y} y' = \phi_y(x, x', s, s', y)$. In other words, the transition relation is extended by updating the auxiliary variables in Y according to both the current and next values of variables in $X \cup S$ and the current values of variables in Y (using the functions ϕ_y). Like Skolem variables, the values stored in auxiliary variables do not alter program behavior: P is safe w.r.t. property φ iff $P[S, Y, \phi]$ is.

Intuitively, the auxiliary variables, also known as *monitors* [21], *ghost variables*, or *spec variables* [22,8,2,18], are additional variables whose values depend on the program state, but do not affect the values of other program variables. The auxiliary state is solely a proof device (*i.e.,* they are not maintained during program execution) and are used to explicate implicit program invariants.

For program $P = \langle X, L, \ell_0, R \rangle$ and predicate ψ over $X \cup X'$, define the ψ-*reduced program* $P_\psi = \langle X, L, \ell_0, R_\psi \rangle$, where $\langle \ell, \rho, \ell' \rangle \in R$ iff $\langle \ell, \rho \wedge \psi, \ell' \rangle \in R_\psi$. An *auxiliary invariant* for $P[S, Y, \phi]$ is a predicate ψ over $X \cup S \cup Y \cup X' \cup S' \cup Y'$ such that the transition relation of $P[S, Y, \phi]$ restricted to the reachable states is a subset of ψ.

Lemma 3. (Auxiliary Invariant) *Let P be a program and φ a set of states of P. For Skolem variables S, auxiliary variables Y, and auxiliary update functions ϕ, if ψ is an auxiliary invariant for $P[S, Y, \phi]$ and $P[S, Y, \phi]_\psi$ is safe w.r.t. φ, then P is safe w.r.t. φ.*

Auxiliary Invariants via Reference Predicates. Even after Skolemization we are left with verification obligations over unbounded state spaces, as the reference counts are unbounded. To solve this problem, we introduce an auxiliary invariant relating a resource's reference count with the number of references to it. A *reference predicate* is a quantifier-free predicate Π over program variables, that is parameterized by two variables, a *source i* and *target j*. A reference predicate Π defines, for each source i, a *reference (target) set*

$$\Pi(i) \doteq \{j \mid \Pi(i, j)\}$$

For each reference predicate, we automatically add auxiliary variables that track the *cardinality* of $\Pi(i)$, by instrumenting the program with an unbounded auxiliary array x_Π that maps the domain of sources to the domain of targets. Assume that: **(A1)** in the initial state, the reference predicate is false for all sources and targets, and, **(A2)** each transition affects only a *finite, named* set of sources and targets. The first assumption is semantic, and depends on the choice of the reference predicate. The second assumption can be syntactically enforced. Under these assumptions, we can automatically instrument the program with auxiliary transitions that (1) initialize $x_\Pi[i]$ with 0, and, (2) increment (resp. decrement) the auxiliary counter $x_\Pi[i]$ whenever for some j, a program transition turns $\Pi(i, j)$ toggles from false to true (resp. from true to false). This auxiliary instrumentation ensures "by construction" the invariant: $x_\Pi[i] = |\Pi(i)|$, *i.e.,* that $x_\Pi[i]$ equals the cardinality of the reference target set $\Pi(i)$. Finally, we instrument the program (*i.e.,* conjoin the set of reachable states) with the auxiliary invariant $\Pi(i, j) \Rightarrow x_\Pi[i] > 0$ for all syntactic sources and targets i and j. This invariant follows from the meta-theorem that if for some i, j, the reference predicate $\Pi(i, j)$ holds, then the reference set $\Pi(i)$ is non-empty, and hence its cardinality $x_\Pi[i]$ is positive.

This strategy addresses the unboundedness of the reference counts as follows. First, it uses a semantic notion of reference (the reference predicate) to instrument the program with a "correct-by-construction" reference counter. Second, in the program reduced w.r.t. the auxiliary invariant, we have replaced the *global* check that the implemented reference count is *positive* with the *local* check that the implemented reference count *equals* the auxiliary reference count. Though the auxiliary counter x_Π is an unbounded array, we can apply case splitting and Skolemization to this array as with the original

program variables. Notice, that strategy only assumes the simple, enforceable, requirements **A1, A2** about the reference counts and program. In particular, it does not assume that the program performs correct reference counting.

Example. The reference relationship for the program in Figure 1 is captured by the predicate:

$$\Pi(\texttt{g}, \texttt{pid}) \doteq (\texttt{ref[pid]=g})$$

which states that there is a reference to the source object $\texttt{g}$ from a target process $\texttt{pid}$ iff $\texttt{ref[pid]=g}$. For this reference predicate, x_Π is the auxiliary correct-by-construction reference count array such that $x_\Pi[\texttt{g}]$ equals the *number* of processes that have a reference to $\texttt{g}$. Our tool automatically instruments the program so that each element of x_Π is initially 0. Further, transitions are added to increment (resp. decrement) an element of x_Π whenever the resource corresponding to the element is acquired (resp. released). This instrumentation is performed by the function $\texttt{update\_aux}$ in Figure 3, which takes as input the counter $x_\Pi[\texttt{g}]$ and the predicate $\texttt{ref[pid]=g}$ and increments (resp. decrements) the counter if the predicate toggles from false to true (resp. true to false) in executing the transition. Figure 3 shows the program instrumented with the case splits induced by the Skolem variables (line 3), the auxiliary variable x_Π, and the auxiliary update function $\texttt{update\_aux}$. Finally, our tool automatically strengthens the program with the auxiliary invariant $\texttt{ref[pid]=g} \Rightarrow x_\Pi[\texttt{g}]\texttt{>0}$ that follows from the instrumentation and the meta-theorem described above. Thus, our technique uses the manually specified reference predicate to instrument the program with correct-by-construction counters, following which the verification task is reduced to proving, that for each $\texttt{g}$, we have $\texttt{count[g]} = x_\Pi[\texttt{g}]$ (which, conjoined with the auxiliary invariant proves the reference count assertion on at Line 3). Finally, note that via Skolemization, the above reduces to proving $\texttt{count[G]} = x_\Pi[\texttt{G}]$ for an arbitrary object $\texttt{G}$.

Step 4: Model Checking
Once we have introduced Skolem variables and auxiliary invariants, we can apply a software model checker such as SLAM or BLAST to discharge the assertion. Algorithm 1 shows a worklist based abstract model checking algorithm using an auxiliary invariant. Its soundness is standard. The procedure PredAbs on Line 8 computes an abstraction of the concrete transition relation relative to an abstract domain (in our implementation, predicate abstraction with transition refinement [11,13,15]). Notice that the abstraction assumes that the auxiliary invariant holds along each transition. (Techniques to automatically find appropriate predicates [5,12] are orthogonal, and can be combined with our algorithm.)

Lemma 4. (Soundness) *If Algorithm 1 returns* SAFE *and ψ is an auxiliary invariant of P, then P is safe w.r.t. φ.*

Example. In our example, the model checker runs a program consisting of an unbounded number of processes each executing the code, where each instruction in the code (each line in the example) is considered atomic. Consider the predicates:

$$\text{(a) } \texttt{pid=P}, \quad \text{(b) } \texttt{g=G}, \quad \text{(c) } \texttt{count[G]>0},$$
$$\text{(d) } \texttt{ref[P]=G}, \quad \text{(e) } x_\Pi[\texttt{G}]\texttt{=count[G]}, \quad \text{(f) } \texttt{count[G]}\geq\texttt{0}.$$

Algorithm 1. Symbolic Model Checking

Input: Program $P = \langle X, L, \ell_0, R \rangle$, States φ, Predicates Π, Auxiliary Invariant ψ
Result: SAFE or UNSAFE
Data: Queue worklist, Incremental per-location invariant η

1   worklist $\leftarrow [\langle \ell_0, true \rangle]$;
2   $\eta \leftarrow \lambda\ell.false$;
3   **while** worklist *is not empty* **do**
4   remove $\langle \ell, \hat{s} \rangle$ from worklist;
5   **if** $\hat{s} \Rightarrow \eta(\ell)$ *is not valid* **then**
6         $\eta \leftarrow \eta[\ell \mapsto \hat{s} \vee \eta(\ell)]$;
7   **foreach** $\langle \ell, \rho, \ell' \rangle \in R$ **do**
8   add $\langle \ell', \mathsf{PredAbs}(\hat{s} \wedge \rho \wedge \psi, \Pi) \rangle$ to worklist;
9   **end**
10   **end**
11 **end**
12 **if** $\forall\ell.\ \eta(\ell) \Rightarrow \varphi(\ell)$ **then return** SAFE **else return** UNSAFE

Predicates (a) and (b) allow us to *strongly* track facts of the "interesting" array indices. (c) and (d) track whether the (arbitrary) process P references a resource with a positive count. Predicate (e) tracks whether the auxiliary and actual counters agree on the reference counts, and (f) is needed to derive (c) when the counts is incremented. These predicates are sufficient for our model checker to synthesize the inductive invariant

$$x_\Pi\texttt{[G]=count[G]} \ \wedge \ (\texttt{ref[P]=G} \Rightarrow \texttt{count[G]>0})$$

describing an over-approximation of the reachable program states (Figure 4 shows this calculation), which suffices to prove the case-split assertion at line 3 of Figure 3. Though the first two predicates do not appear in the invariant, they are essential for its derivation as they enable strong updates on the Skolemized cells of count, ref, and x_Π.

Temporal case splitting on the Skolem variables reduces the infinite number of processes and resources to a finite set. Similarly, the auxiliary invariant and counter ensure that we need only to track the relationship between the auxiliary and actual counters, and whether the former is positive, instead of precisely tracking an unbounded counter. It is the combination of these techniques that allows the model checker to prove such a complex property of an unbounded system.

Example: Buggy Reference Counting. Figure 2 shows a reference count implementation that is contains a bug that arises from aliasing. The program is motivated by an actual bug in an implementation of the Python language [28]. Each client works atomically in a loop, acquiring a new resource and releasing the old resource in each iteration. The error occurs when the old resource in item is the same as the new resource in new_item, and the resource has reference count of 1 (*i.e.,* the client holds the only reference to this resource). The decref on line 9 then decreases the reference count to 0, and frees the resource. However, the client still holds a reference to the same resource in new_item (from line 8), so the incref at line 11 erroneously writes to freed memory, possibly corrupting it. Our tool does find an error trace for this buggy

location	abstract states			
scheduler	*true*			
1	a		$\bar{a}$	
2	ab	$a\bar{b}$	$\bar{a}b$	$\bar{a}\bar{b}$
3	$abcd$	$a\bar{b}\bar{d}$	$\bar{a}bc$	$\bar{a}\bar{b}$
4	$abcd$	$a\bar{b}\bar{d}$	$\bar{a}bc$	$\bar{a}\bar{b}$
5	$abcd$	$a\bar{b}\bar{d}$	$\bar{a}bc$	$\bar{a}\bar{b}$
exit	$ab\bar{d}$	$a\bar{b}\bar{d}$	$\bar{a}b$	$\bar{a}\bar{b}$

invariant: ef

Fig. 4. The reachable abstract program states of the program in Figure 3 at each program location w.r.t. the predicates (a)–(f) given above. A string of (possibly barred) predicates indicates a partial valuation where each non-barred predicate is true, each barred predicate is false, and each unmentioned predicate may be either true or false. (Conceptually the partial valuation is a disjunction of (total) valuations.) The predicates (e) and (f) hold universally, and the string ef is implicitly appended to each valuation. The scheduler is responsible for deciding which process pid executes.

program. Furthermore, our technique is able to prove safe the correct version of the program, where the reference count is incremented before the decrement on line 9.

Limitations. One limitation of our approach is that the Skolem variables necessary for verification are not mechanically determined; this is left for the user of our analysis tool. In our experience with reference counting we have found the number to be small (one, or two, per structure) and easy to find, but the search for appropriate Skolems can be hard in general. Second, our approach is constrained by the invariants that can be expressed by the abstract domain, and the design of an appropriate domain can be hard for complicated invariants, especially with rich quantifier structures. Our technique only checks that when a resource is accessed, it has a positive reference count. This property by itself does not guarantee the absence of memory leaks, for example, those caused by cyclic structures of references that are not reachable from any program variable.

3　Case Studies

In addition to the simple examples from Section 2, we have applied our tool to two case studies of reference counting in real systems code: a page allocator derived from the JOS kernel [17], and the YAFFS file system [23].

We use a *logical memory model*: memory is represented as an unbounded array Mem of elements large enough to hold any structure allocated in the program. Each memory cell is annotated with a valid bit, initially each with value 0. Pointers are modeled as indices to the Mem array, and index 0 denotes null. Our implementation of malloc nondeterministically chooses an index i such that Mem[i].valid = 0, sets Mem[i].valid to 1, and returns i. Our implementation of free(i) ensures that Mem[i].valid = 1, and resets Mem[i].valid to 0.

We model concurrency by calling the top-level procedures, which are considered atomic, inside of a loop which nondeterministically chooses a process/thread identifier pid and a procedure proc and executes proc as pid.

```
typedef struct env {
    int env_mypp;
    int env_pgdir[NVPAGES];
    struct env *env_prev;
    struct env *env_next;
} env_t;

int pages[NPPAGES];
int page_protected[NPPAGES];
env_t *envs = NULL;
```

Fig. 5. Environment data structures in JOS

```
int page_alloc(env_t *env, int vp) {
    int pp = page_getfree();
    if (pp < 0) return -1;
    if (env->env_pgdir[vp] >= 0)
        pages[ env->env_pgdir[vp] ]--;
    env->env_pgdir[vp] = pp;
    pages[pp]++;
    return 0;
}
int page_unmap(env_t *env, int vp) {
    if (env->env_pgdir[vp] >= 0) {
        pages[ env->env_pgdir[vp] ]--;
        env->env_pgdir[vp] = -1;
    }
}
int page_map(env_t *srcenv, int srcvp,
             env_t *dstenv, int dstvp) {
  if (srcenv->env_pgdir[srcvp] < 0)
        return -1;
  pages[ srcenv->env_pgdir[srcvp] ]++;
  if (dstenv->env_pgdir[dstvp] >= 0)
      pages[ dstenv->env_pgdir[dstvp] ]--;
  dstenv->env_pgdir[dstvp] =
      srcenv->env_pgdir[srcvp];
  return 0;
}
```

Fig. 6. Page directory manipulation in JOS. `page_getfree` returns the index to an unused page, if one is exists, and -1 otherwise.

```
env_t *env_alloc(void) {
 env_t *env;
 int i, env_pp = page_getfree();
 if (env_pp < 0) return NULL;
 env = (env_t *) malloc(sizeof(env_t));
 env->env_mypp = env_pp;
 for (i = 0; i < NVPAGES; i++)
     env->env_pgdir[i] = -1;

 /* put on list */
 env->env_next = envs;
 env->env_prev = NULL;
 if (envs) envs->env_prev = env;
 envs = env;
 pages[env_pp]++;
 page_protected[env_pp] = 1;
 return env;
}

void env_free(env_t *env) {
 int i;
 for (i = 0; i < NVPAGES; i++)
     if (env->env_pgdir[i] >= 0)
         pages[ env->env_pgdir[i] ]--;
 page_protected[env->env_mypp] = 0;
 pages[ env->env_mypp ]--;

 /* take off list */
 if (env->env_next)
     env->env_next->env_prev =
         env->env_prev;
 if (env->env_prev)
     env->env_prev->env_next =
         env->env_next;
 else envs = env->env_next;
 free(env);
}
```

Fig. 7. Environment (de)allocation in JOS

JOS Memory Mapping. In JOS [17], a simple operating system used as an educational aid, memory is organized as an array of physical pages, to which user processes (or *environments*) hold virtual page mappings (see Figures 5–7). The environment structure (`env_t`) stores the index of a protected physical page (`env_mypp`), a virtual page table (`env_pgdir`), and pointers used for the kernel's doubly linked list of environments (`env_prev`, `env_next`). The `pages` array maintains the number of virtual page mappings to each physical page, or 1 for protected pages (*i.e.,* the `env_mypp` of some environment, explicitly marked by the `page_protected` array). The kernel ensures that an `env_pgdir` entry is not protected.

To verify that every live `env_pgdir` entry has a positive reference count we introduce: a single physical page Skolem variable, one auxiliary counter variable for each page, and an auxiliary invariant insisting mapped pages' auxiliary counters are positive. Model checking ensures the auxiliary counters are equal to JOS's reference counters.

```
int yaffs_open(...) {                    void yaffs_close(...) {
    ...                                      ...
    h = yaffsfs_GetHandlePointer(...);       h = yaffsfs_GetHandlePointer(...);
    obj = yaffsfs_FindObject(...);           if (h && h->inUse) {
    ...                                          h->obj->inUse--;
    h->obj = obj; obj->inUse++;                  if (h->obj->inUse <= 0)
    ...                                              yaffs_DeleteFile(h->obj);
}                                                h->obj = 0;
                                             }
                                         }
```

Fig. 8. YAFFS reference counting, simplified

Given the Skolems and the auxiliary invariant, our tool proves the correct use of reference counts for any number of pages and environments. (Memory leak freedom is proved by ensuring the values of the actual and auxiliary counters coincide.) The reachability analysis requires 17 predicates and 29 seconds.

Yaffs File Object Management. The YAFFS log-structured filesystem for flash memory [23] represents files with heap-allocated `yaffs_Object` structures, each containing a reference counting `inUse` field (Figure 8 shows fragments of a simplified version, although we have verified the actual implementation). Users access objects indirectly through the `obj` field of a `yaffs_Handle` pointer. The handles are stored in a fixed-sized array, indexed by an integer file handle descriptor.

File read and write operations access `yaffs_Object`s under the assumption that their reference counts are positive. To verify, we introduce a single file object Skolem variable. As done for JOS, we also introduce auxiliary state to track handle-object (un)mappings, and equate that state with actual object reference counts by symbolic model checking. Assuming that each file operation occurs atomically, our tool is able to prove the sound use of reference counts for any number of handles and objects. The reachability analysis requires 34 predicates and 36 seconds.

4   Related Work

Compositional Verification. Our use of temporal case splitting with Skolem variables is inspired by similar approaches in hardware verification [22], where a hardware design is decomposed into units of work and the finite instantiations verified using a BDD-based model checker. Our work differs from the above in two respects. First, we consider C programs where heap locations are allocated dynamically, and need not have static names. Second, by using predicate abstraction over more expressive theories (*e.g.,* equality, arithmetic, arrays) we may track relationships between variables, which is generally required to prove sound reference counting.

A restricted use of Skolem variables to separate a safety verification problem into sub-problems has been suggested before [30,4]. However, an analysis with a dataflow analysis back-end [30] merges states at join points, and cannot perform case splits over the abstract domain used in our examples. There the benefits of separation were restricted to syntactically disjoint choices (for example, where there were separate assertions on two arrays, and the abstraction would first prove the assertion for the first array while abstracting the second, and then prove it separately for the second). We, on

the other hand, perform case splitting on the temporal behavior of the program, thus correlating the choice of a Skolem at one point in the execution to a subsequent check.

Predicate Abstraction. Skolem variables have been used with predicate abstraction to infer universally quantified invariants over the program state [9,19,20]. However, the properties considered thus far have been limited, for the most part, to simple intraprocedural reasoning about arrays. We believe that one reason for this is that fast Cartesian predicate abstraction, implemented as the default in software model checkers such as SLAM or BLAST, is too coarse for reasoning about array and pointer variables. Our work builds on the interpolant-based transition relation refinement [15] that lazily refines the Cartesian abstraction to the required precision, and our reachability engine uses this refinement for scalability. The use of quantified predicates in model checking based on predicate abstraction [27,7] has, for similar reasons, been limited to small and abstract encodings of complicated procedures (*e.g.,* garbage collectors). In these applications quantifiers are instantiated by matching heuristics implemented in the theorem prover, or manually. In contrast, our use of Skolems, while less powerful than full quantification, is more predictable, and does not rely on matching heuristics. While we concentrate on the technique of using predicate abstraction with Skolems and auxiliary state in reachability analysis, techniques to *infer* quantified predicates [20] are orthogonal, and can be combined with our algorithm to find such predicates. Finally, auxiliary invariants have been used in software model checking, *e.g.,* to approximate the shape of the heap using alias analysis [1,13], or to infer polyhedral invariants in a prepass before applying predicate abstraction [14].

Shape Analysis. Shape analysis [26] and separation logic [25,3] are powerful frameworks for reasoning about heap manipulating programs. While our techniques can be simulated inside shape analysis, our advantage is the use of already developed efficient and scalable predicate abstraction and manipulation engines (from BLAST) to reason about heap properties on large programs. Shape analysis has also been used to verify concurrent programs with an unbounded number of threads through the use of manually supplied instrumentation predicates [29]. Skolemization and case splitting are orthogonal—once they are performed, three valued logic based analyses can be used to discharge the reduced model checking tasks.

Work on canonical abstraction of arrays [10,16] is close to our work: there, an (unbounded) array is abstracted with respect to an iterator into the portion of the array before, at, and after the iterator; these portions are summarized with respect to the predicates that hold on them. In contrast, our technique abstracts an array into the locations indexed by Skolems, and all other locations; additional refinements are introduced with additional Skolems and predicate relationships between values at the Skolem indices. Instead of a specialized dataflow analysis, we perform path-sensitive model checking that can then correlate data at Skolem locations. Our experience is that for properties that depend on an arbitrary element of the array, Skolemization and case splitting provides a more natural (and often, a more succinct) abstraction of the program.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
2. 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)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
4. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
7. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: FMCAD (2002)
8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI (2002)
9. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL (2002)
10. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: POPL (2005)
11. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
12. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
14. Jain, H., Ivančić, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 137–151. Springer, Heidelberg (2006)
15. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
16. 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)
17. Jos, J.: An operating system kernel,
    http://pdos.csail.mit.edu/6.828/2005/overview.html
18. Kuncak, V., Lam, P., Zee, K., Rinard, M.C.: Modular pluggable analyses for data structure consistency. IEEE Trans. Software Eng. 32(12), 988–1005 (2006)
19. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 267–281. Springer, Heidelberg (2004)
20. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
21. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Heidelberg (1995)

22. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Sci. Comput. Program. 37, 279–309 (2000)
23. One, A.: Yaffs file system, `http://www.yaffs.net/`
24. Qadeer, S., Wu, D.: KISS: Keep it simple, sequential. In: PLDI (2004)
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
26. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL (1999)
27. Shankar, N.: Combining theorem proving and model checking through symbolic analysis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, p. 1. Springer, Heidelberg (2000)
28. van Rossum, G.: Debugging reference count problems, `http://www.python.org/doc/essays/refcnt/`
29. Yahav, E.: Verifying safety properties of concurrent java programs using 3-valued logic. In: POPL (2001)
30. Yahav, E., Ramalingam, G.: Verifying safety properties using separation and heterogeneous abstractions. In: PLDI (2004)

Falsification of LTL Safety Properties in Hybrid Systems

Erion Plaku, Lydia E. Kavraki, and Moshe Y. Vardi

Dept. of Computer Science, Rice University, Houston TX 77005
{plakue,kavraki,vardi}@cs.rice.edu

Abstract. This paper develops a novel computational method for the falsification of safety properties specified by syntactically safe linear temporal logic (LTL) formulas ϕ for hybrid systems with general nonlinear dynamics and input controls. The method is based on an effective combination of robot motion planning and model checking. Experiments on a hybrid robotic system benchmark with nonlinear dynamics show significant speedup over related work. The experiments also indicate significant speedup when using minimized DFA instead of non-minimized NFA, as obtained by standard tools, for representing the violating prefixes of ϕ.

1 Introduction

Hybrid systems, which combine discrete and continuous dynamics, provide sophisticated mathematical models being used in robotics, automated highway systems, air-traffic management, computational biology, and other areas [1]. An important problem in hybrid systems is the verification of safety properties [1, 2], which assert that nothing "bad" happens, e.g., "the car avoids obstacles." A hybrid system is safe if there are no witness trajectories indicating a safety violation. Safety properties have traditionally been specified in terms of a set of unsafe states and verification has been formulated as reachability analysis [1, 2, 3, 4, 5, 6, 7]. Reachability analysis in hybrid systems is in general undecidable [2, 3]. Moreover, complete algorithms have an exponential dependency on the dimension of the state space and are limited in practicality to low-dimensional systems [1, 2, 4].

To handle more complex hybrid systems, alternative methods[8, 9, 10, 11, 12] have been proposed that shift from verification to falsification, which is often the focus of model checking in industrial applications [13]. Even though they are unable to determine that a system is safe, these methods may compute witness trajectories when the system is not safe. Witness trajectories, similar to error traces in model checking[13], indicate modeling flaws, which designers can then correct. The falsification methods in [8, 9, 10] adapt the Rapidly-exploring Random Tree (`RRT`) motion planner [14], which was originally developed for reachability analysis in continuous systems. We recently proposed the Hybrid Discrete Continuous Exploration (`HyDICE`) falsification method [11, 12], which also takes advantage of motion planning, but shows significant speedup over related work [9, 10].

As more complex hybrid systems are considered, limiting safety properties to a set of unsafe states, as in current methods [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],

considerably restricts the ability of designers to adequately express the desired safe behavior of the system. To allow for more sophisticated properties, researchers have advocated the use of linear temporal logic (LTL), which makes it possible to express safety properties with respect to time, such as "if the concentration level of gene A reaches x, then the concentration level of gene B will never reach y." LTL has been widely used in model checking of discrete systems in software and hardware [15], and timed systems [16]. The work in [17] generated trajectories that satisfy LTL constraints on the sequence of triangles visited by a point robot with Newtonian dynamics by using a controller that could drive the robot between adjacent triangles. The work in [18] used LTL to analyze gene networks. The work in [19] developed a method to verify LTL safety properties for robust discrete-time hybrid systems.

Traditional approaches for verification of an LTL property ϕ on a hybrid system $\mathcal{H}$ often cast the problem as reachability analysis via model checking. Abstractions are typically used to obtain a discrete transition model $\mathcal{M}$ that simulates $\mathcal{H}$, so that checking ϕ on $\mathcal{M}$ is sufficient to checking ϕ on $\mathcal{H}$ [5]. Moreover, with an exponential blow-up at most, a nondeterministic finite automaton (NFA) $\mathcal{A}$ can be constructed that describes all prefixes violating ϕ [20]. This allows for checking ϕ on $\mathcal{H}$ via model checking on $\mathcal{M} \times \mathcal{A}$. The challenge lies in the computation of $\mathcal{M}$, which is limited in practicality to low-dimensional hybrid systems due to the exponential dependency on the state-space dimension [1, 2, 4].

Applying alternative approaches [8, 9, 10, 11, 12] to falsify LTL safety properties by reachability analysis is also challenging due to intricacies of motion planning. During the search, motion planning extends a tree $\mathcal{T}$ in the state space of $\mathcal{H}$ by adding valid trajectories as new branches. Consider a vertex v and the trajectory ζ from the root of $\mathcal{T}$ to v. In reachability analysis [8, 9, 10, 11, 12], a witness trajectory is found when the state associated with v is unsafe. When considering LTL, such criteria is not sufficient, since ζ needs to satisfy $\neg\phi$. It then becomes necessary to maintain the propositional assignments satisfied by ζ and to effectively extend $\mathcal{T}$ so that more and more of the propositional assignments of $\neg\phi$ are satisfied.

To handle LTL, one can consider a naïve extension of the work in [8, 9, 10, 11, 12] by using $\mathcal{A}$ as an external monitor to determine when a tree trajectory ζ satisfies $\neg\phi$ by keeping track of the automaton states associated with each ζ. As shown in this work, however, such an approach is computationally very inefficient.

The main contribution of this work is to extend `HyDICE` [11, 12] in order to effectively incorporate LTL safety properties into hybrid-system falsification. The proposed approach, termed `TemporalHyDICE`, can be used to compute witness trajectories for the falsification of properties specified by syntactically safe LTL formulas for hybrid systems with

– external inputs, which could represent controls, uncertainties; and
– general nonlinear dynamics, where $\textsc{Flow}_q(x, u, t)$ is treated as a black box that outputs a state x_{new} obtained by following the hybrid-system dynamics when at state (q, x) and applying the input u for t time units.

When differential equations describe the dynamics, closed-form solutions (if available) or numerical integrations can be used for the black-box simulation.

When differential equations become too cumbersome to describe the dynamics, other computer programs can be used for the simulation.

In its core, `TemporalHyDICE` draws from research in traditional and alternative approaches in hybrid systems to synergistically combine model checking and motion planning. This combination presents significant challenges, as it requires dealing with important issues, such as state-space search, memory usage, scalability, and passing of information between model checking and motion planning. In `TemporalHyDICE`, model checking guides motion planning by providing feasible directions along which to extend $\mathcal{T}$. A feasible direction consists of a sequence $[\tau_i]_{i=1}^n$ of propositional assignments that violates ϕ, which is computed by searching on-the-fly a discrete transition model $\mathcal{M}$ of $\mathcal{H}$ and the automaton $\mathcal{A}$ of $\neg\phi$. By not computing $\mathcal{M} \times \mathcal{A}$ explicitly, `TemporalHyDICE` considerably reduces the memory used by model checking. Moreover, unlike traditional approaches, `TemporalHyDICE` does not require $\mathcal{M}$ to simulate $\mathcal{H}$. In fact, $\mathcal{M}$ is based on a simple partition of the state space of $\mathcal{H}$ induced by propositions in ϕ. Motion planning extends $\mathcal{T}$ along directions $[\tau_i]_{i=1}^n$ provided by model checking so that more and more of $\tau_1, \ldots, \tau_n$ are satisfied in succession. As motion planning extends $\mathcal{T}$, it also gathers information to estimate the progress made in the search for a witness trajectory. This information is fed back to model checking to select in future iterations increasingly feasible directions for extending $\mathcal{T}$. This interactive combination of model checking and motion planning is a crucial component that allows `TemporalHyDICE` to effectively search for a witness trajectory.

An initial validation of `TemporalHyDICE` is provided by falsifying many properties specified by syntactically safe LTL formulas for a nonlinear hybrid robotic system. Experiments show significant speedup over related work. This work also studies the impact of representing $\neg\phi$ by DFAs or NFAs, as obtained by standard tools. The motivation comes from the work in [21], which shows significant speedup when using DFAs instead of NFAs in model checking. Experiments in this work in the context of falsification of LTL safety properties in hybrid systems also indicate significant speedup when using DFAs instead of NFAs.

The rest is as follows. Section 2 contains preliminaries. A straightforward approach of incorporating LTL into related work [8, 9, 10, 11, 12] by using the automaton $\mathcal{A}$ as an external monitor is described in Section 3. As demonstrated by the experiments, such an approach, however, is computationally very inefficient. The proposed approach, `TemporalHyDICE`, which effectively incorporates LTL into hybrid-system falsification, is described in Section 4. Experiments and results are described in Section 5. The paper concludes in Section 6 with a discussion.

2   Preliminaries

This section defines hybrid automata, LTL, the automata for the complement of LTL formulas, and the problem statement.

Hybrid Systems: Hybrid systems are modeled by hybrid automata [2]. A hybrid automaton is a tuple $\mathcal{H} = (S, I, \text{INV}, E, \text{GUARD}, \text{JUMP}, U, \text{FLOW})$, where $S = Q \times X$ is a product of a discrete and finite set Q and continuous spaces $X = \{X_q : q \in Q\}$; $I \subset S$ denotes initial states; $\text{INV} = \{\text{INV}_q : q \in Q\}$, where $\text{INV}_q :$

$X_q \rightarrow \{\top, \bot\}$ is the invariant function; $E \subseteq Q \times Q$ denotes discrete transitions; $\text{GUARD} = \{\text{GUARD}_{q_i,q_j} : (q_i, q_j) \in E\}$ and $\text{JUMP} = \{\text{JUMP}_{q_i,q_j} : (q_i, q_j) \in E\}$, where $\text{GUARD}_{q_i,q_j} : X_{q_i} \rightarrow \{\top, \bot\}$ and $\text{JUMP}_{q_i,q_j} : X_{q_i} \rightarrow X_{q_j}$ denote guard and jump functions, respectively; $U = \{U_q : q \in Q\}$, where an input in $U_q \subseteq \mathbb{R}^{\dim(U_q)}$ can represent controls, nondeterminism, or uncertainties; and $\text{FLOW} = \{\text{FLOW}_q : q \in Q\}$, where $\text{FLOW}_q : X_q \times U_q \times \mathbb{R}^{\geq 0} \rightarrow X_q$ is the flow function. This work treats the dynamics as a black box, where $\text{FLOW}_q(x, u, t)$ outputs the state obtained by following the dynamics from x when u is applied for t time units. This allows for general nonlinear dynamics. In fact, the only requirement is the ability to simulate the dynamics. $\text{INV}_q : X_q \rightarrow \{\top, \bot\}$, $\text{GUARD}_{q_i,q_j} : X_{q_i} \rightarrow \{\top, \bot\}$, and $\text{JUMP}_{q_i,q_j} : X_{q_i} \rightarrow X_{q_j}$ are also treated as black boxes to allow general specifications that do not limit designers to a particular approach, such as polyhedral or ellipsoidal constraints. A hybrid-system trajectory consists of continuous trajectories interleaved with discrete transitions.

Continuous Trajectory: $s = (q, x) \in S$, $T \geq 0$, $u \in U_q$ define a continuous trajectory $\Psi_{s,u,T} : [0, T] \rightarrow X_q$, where $\Psi_{s,u,T}(t) = \text{FLOW}_q(x, u, t)$, $t \in [0, T]$.

Discrete Transition: For any $(q, x) \in S$, let $\chi(q, x) = (q', \text{JUMP}_{q,q'}(x))$ if $\text{GUARD}_{q,q'}(x) = \top$ for some $(q, q') \in E$. Otherwise, let $\chi(q, x) = (q, x)$.

Continuous Trajectory + Discrete Transition: $\Upsilon_{s,u,T} : [0, T] \rightarrow S$, defined as $\Upsilon_{s,u,T}(t) = (q, \Psi_{s,u,T}(t))$, $0 \leq t < T$ and $\Upsilon_{s,u,T}(T) = \chi(q, \Psi_{s,u,T}(T))$, ensures that a discrete transition at time T, if it occurs, is followed.

Trajectory Extension: Extending $\Phi : [0, T] \rightarrow S$ by applying $u' \in U$ to $\Phi(T)$ for $T' \geq 0$ time units, written as $\Phi \circ (u', T')$, is a trajectory $\Xi : [0, T + T'] \rightarrow S$ where $\Xi(t) = \Phi(t), t \in [0, T]$ and $\Xi(t) = \Upsilon_{\Phi(T),u',T'}(t - T), t \in (T, T + T']$.

Hybrid-System Trajectory: A state $s \in S$, a sequence $u_1, \dots, u_k$ of inputs, and a sequence $T_1, \dots, T_k$ of times define a trajectory $\zeta : [0, T] \rightarrow S$, where $T = T_1 + \dots + T_k$ and $\zeta = \Upsilon_{s,u_1,T_1} \circ (u_2, T_2) \circ \dots \circ (u_k, T_k)$.

In this work, a discrete transition is taken when a guard condition is satisfied. There is, however, no inherent limitation in dealing with non-urgent discrete transitions. In such cases, enabled discrete transitions could be taken nondeterministically or taken only when the invariant is invalid or a combination of both.

LTL: Let Π denote a set of propositional variables.

LTL Syntax and Semantics [20]: Every $\pi \in \Pi$ is a formula. If ϕ and ψ are formulas, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\mathcal{X}\phi$ (next), $\phi\mathcal{U}\psi$ (until), $\phi\mathcal{R}\psi$ (release), $\mathcal{F}\phi$ (future), and $\mathcal{G}\phi$ (globally) are also formulas. Let $\sigma = \tau_0, \tau_1, \dots \in 2^{\Pi}$. Let $\sigma^i = \tau_i, \tau_{i+1}, \dots$ We write $\sigma \models \phi$ to indicate that σ satisfies ϕ and define it as $\sigma \models \top$; $\sigma \not\models \bot$; $\sigma \models \pi$ if $\pi \in \tau_0$; $\sigma \models \phi \wedge \psi$ if $\sigma \models \phi$ and $\sigma \models \psi$; $\sigma \models \mathcal{X}\phi$ if $\sigma^1 \models \phi$; $\sigma \models \phi\mathcal{U}\psi$ if $\exists k \geq 0$ s.t. $\sigma^k \models \psi$ and $\forall 0 \leq i < k : \sigma^i \models \phi$; $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$; $\mathcal{F}\phi \equiv \top\mathcal{U}\phi$; $\mathcal{G}\phi \equiv \neg\mathcal{F}\neg\phi$; $\phi\mathcal{R}\psi \equiv \neg(\neg\phi\mathcal{U}\neg\psi)$.

Syntactically Safe LTL [22]: An LTL formula ϕ that, when written in positive normal form, uses only the temporals $\mathcal{X}$, $\mathcal{R}$, and $\mathcal{G}$ is syntactically safe. Every syntactically safe formula is a safety formula.

NFA for Syntactically Safe LTL [20]: With an exponential blow-up at most, an NFA can be constructed that describes all prefixes violating a syntactically safe LTL formula. The NFA is a tuple $\mathcal{A} = (Z, \Sigma, \delta, v_0, \text{Acc})$, where Z is a finite

set of states; $\Sigma = 2^\Pi$ is the input alphabet; $\delta : Z \times \Sigma \to 2^Z$ is the transition function; $z_0 \in Z$ is the initial state; and $\mathrm{Acc} \subseteq Z$ is the set of accepting states. The set of states on which $[\tau_i]_{i=1}^n$, $\tau_i \in 2^\Pi$, ends up when run on $\mathcal{A}$ is defined as

$$\mathcal{A}([\tau_i]_{i=1}^n) = \begin{cases} \delta(z_0, \tau_1), & n = 1 \\ \bigcup_{z \in \mathcal{A}([\tau_i]_{i=1}^{n-1})} \delta(z, \tau_n), & n > 1. \end{cases} \quad \mathcal{A} \text{ accepts } [\tau_i]_{i=1}^n \text{ iff } \mathcal{A}([\tau_i]_{i=1}^n) \cap \mathrm{Acc} \neq \emptyset.$$

LTL over Hybrid-System Trajectories: Let $\Pi = \{\pi_{q,i} : q \in \mathcal{H}.Q \wedge 1 \leq i \leq n_q\}$, where n_q is the number of propositional variables associated with q. The truth-value of each $\pi_{q,i}$ is determined by a black-box function $\mathrm{PROP}_{q,i} : \mathcal{H}.X_q \to \{\top, \bot\}$. The map $\tau : \mathcal{H}.S \to 2^\Pi$ maps $(q, x) \in \mathcal{H}.S$ to truth propositions: $\tau((q, x)) = \{\pi_{q,i} : \pi_{q,i} \in \Pi \text{ and } \mathrm{PROP}_{q,i}(x) = \top\}$. When interpreted over a hybrid-system trajectory ζ, the notation $\tau(\zeta)$ denotes the sequence of propositional assignments $[\tau_i]_{i=1}^n$ ($\tau_i \in 2^\Pi$, $\tau_i \neq \tau_{i+1}$) in the order satisfied by ζ, i.e., $\tau_i = \tau(\zeta(T_i))$ where $0 \leq T_1 < \cdots < T_n \leq |\zeta|$ such that n is as large as possible and $\tau_i \neq \tau_{i+1}$, $1 \leq i < n$. Then, ζ satisfies ϕ, written $\zeta \models \phi$, iff $\tau(\zeta) \models \phi$.

Problem Statement: Let $\mathcal{P} = (\mathcal{H}, \mathcal{A}, \tau)$, where $\mathcal{H}$ is a hybrid automaton; $\mathcal{A}$ is an automaton for the complement of a syntactically safe LTL formula ϕ over propositions Π; and τ is a propositional map interpreted both over hybrid-system states and trajectories. Given $\mathcal{P}$, compute a valid trajectory $\zeta : [0, T] \to \mathcal{H}.S$ that satisfies $\neg \phi$, i.e., $(\forall t \in [0, T] : \mathrm{INV}_{q_t}(x_t) = \top$, where $(q_t, x_t) = \zeta(t))$ and $\zeta \models \neg\phi$.

3   Incorporating LTL into Motion-Planning Approaches

Motion planning has been widely used in reachability analysis for continuous robotic systems with dynamics [23, 24]. These methods rely on a common framework that iteratively extends a tree in the state space of the system by adding valid trajectories as branches. Recently, the work in [8, 9, 10] adapted the tree-search framework for reachability analysis in hybrid systems.

There have been no discussions in the literature on how to augment the tree-search framework with LTL trajectory properties, cf. [8, 9, 10]. This section describes a minimal extension of the tree-search framework to handle LTL. The idea is to use $\mathcal{A}$ (DFA or NFA) to keep track of the automaton states associated with each tree trajectory and to determine when a tree trajectory is a witness. In this way, similar to model checking, the tree-search framework searches on-the-fly $\mathcal{H}$ and $\mathcal{A}$. With these modifications, the tree-search framework can be used to falsify LTL safety properties in hybrid systems, and, thus, provide a basis for the experimental comparisons. As demonstrated by the experiments, such an approach, however, is computationally very inefficient. Section 4, which describes `TemporalHyDICE`, then shows how to effectively combine the LTL tree-search framework with model checking on $\mathcal{M}$ and $\mathcal{A}$, where $\mathcal{M}$ is a discrete transition model of $\mathcal{H}$, in order to significantly increase its computational efficiency.

Incorporating LTL into the Tree-Search Framework: The tree is maintained as a graph $\mathcal{T} = (V, E)$. Each vertex $v \in \mathcal{T}.V$ is associated with a state $s \in \mathcal{H}.S$, written as $v.s$. An edge $(v', v'') \in \mathcal{T}.E$ indicates that a valid trajectory connects $v'.s$ to $v''.s$. As the search proceeds iteratively, $\mathcal{T}$ is extended by

Algorithm 3.1. LTL-TSF: Incorporating LTL into the Tree-Search Framework

> **Input:** $\mathcal{P}$: problem specification; $t_{\max} \in \mathbb{R}^{>0}$: upper bound on computation time
> **Output:** A solution trajectory if one is found or $\bot$ otherwise

(a) $\mathcal{T} \leftarrow$ INITALIZETREE($\mathcal{P}$)
while ELAPSEDTIME $< t_{\max}$ **do**
  (b) $v \leftarrow$ SELECTVERTEXFROMTREE($\mathcal{P}, \mathcal{T}$) $\diamond$ *varies from method to method*
  (c) $[u, T, s_{\text{new}}, \alpha_{\text{new}}] \leftarrow$ EXTENDTREE($\mathcal{P}, \mathcal{T}, v$)
  (d) **if** $T > 0 \wedge |\alpha_{\text{new}}| > 0$ **then** $v_{\text{new}} \leftarrow$ ADDBRANCHTOTREE($\mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}]$)
  (e) **if** $\mathcal{P}.\mathcal{A}.\text{Acc} \cap \alpha_{\text{new}} \neq \emptyset$ **then return** TRAJ($\mathcal{T}, v_{\text{new}}$)
return $\bot$

EXTENDTREE($\mathcal{P}, \mathcal{T}, v$) :=
1: $\epsilon \in \mathbb{R}^{>0} \leftarrow$ time step; $n_{\text{steps}} \in \mathbb{N} \leftarrow$ number of steps
2: $s = (q, x) \leftarrow v.s$; $\alpha \leftarrow v.\alpha$; $x_0 \leftarrow x$; $\alpha_0 \leftarrow \alpha$; $\tau_0 \leftarrow \mathcal{P}.\tau(s)$
3: $u \leftarrow$ sample control from $\mathcal{P}.\mathcal{H}.U_q$
4: **for** $i = 1, 2, \ldots, n_{\text{steps}}$ **do** $\diamond$*simulate the continuous and discrete dynamics of* $\mathcal{P}.\mathcal{H}$
5:     $x_i \leftarrow \mathcal{P}.\mathcal{H}.\text{FLOW}_q(x_{i-1}, u, \epsilon)$; $\tau_i \leftarrow \mathcal{P}.\tau((q, x_i))$
6:     **if** $\tau_{i-1} = \tau_i$ **then** $\alpha_i \leftarrow \alpha_{i-1}$ **else** $\alpha_i \leftarrow \cup_{z \in \alpha_{i-1}} \mathcal{P}.\mathcal{A}.\delta(z, \tau_i)$
7:     **if** $\mathcal{P}.\mathcal{H}.\text{INV}_q(x_i) = \bot$ **then return** $[u, (i-1) * \epsilon, (q, x_{i-1}), \alpha_{i-1}]$
8:     **if** $\mathcal{P}.\mathcal{H}.\text{GUARD}_{q, q_{\text{new}}}(x_i) = \top, (q, q_{\text{new}}) \in \mathcal{P}.\mathcal{H}.E$ **then**
9:       $(x_{\text{loc}}, T) \leftarrow$ localize discrete event in $((i-1) * \epsilon, i * \epsilon]$; $\tau_{\text{loc}} \leftarrow \mathcal{P}.\tau((q, x_{\text{loc}}))$
10:      **if** $\tau_{i-1} = \tau_{\text{loc}}$ **then** $\alpha_{\text{loc}} \leftarrow \alpha_{i-1}$ **else** $\alpha_{\text{loc}} \leftarrow \cup_{z \in \alpha_{i-1}} \mathcal{P}.\mathcal{A}.\delta(z, \tau_{\text{loc}})$
11:      $x_{\text{new}} \leftarrow \mathcal{P}.\mathcal{H}.\text{JUMP}_{q, q_{\text{new}}}(x_{\text{loc}})$; $\tau_{\text{new}} \leftarrow \mathcal{P}.\tau((q_{\text{new}}, x_{\text{new}}))$
12:      **if** $\tau_{\text{loc}} = \tau_{\text{new}}$ **then** $\alpha_{\text{new}} \leftarrow \alpha_{\text{loc}}$ **else** $\alpha_{\text{new}} \leftarrow \cup_{z \in \alpha_{\text{loc}}} \mathcal{P}.\mathcal{A}.\delta(z, \tau_{\text{new}})$
13:      **return** $[u, T, (q_{\text{new}}, x_{\text{new}}), \alpha_{\text{new}}]$
14: **return** $[u, n_{\text{steps}} * \epsilon, (q, x_{n_{\text{steps}}}), \alpha_{n_{\text{steps}}}]$

adding new vertices and edges. Consider the trajectory TRAJ($\mathcal{T}, v$) from the root of $\mathcal{T}$ to $v \in \mathcal{T}.V$. If TRAJ($\mathcal{T}, v$) $\models \neg\phi$, then TRAJ($\mathcal{T}, v$) is a witness. To determine TRAJ($\mathcal{T}, v$) $\models \neg\phi$, v is associated with the automaton states corresponding to TRAJ($\mathcal{T}, v$), written as $v.\alpha$ and defined as $v.\alpha = \mathcal{A}(\tau(\text{TRAJ}(\mathcal{T}, v)))$. Then, TRAJ($\mathcal{T}, v$) $\models \neg\phi$ iff $\mathcal{A}(v.\alpha) \cap \mathcal{A}.\text{Acc} \neq \emptyset$. Pseudocode is given in Algo. 3.1.

  (a) InitializeTree($\mathcal{P}$) associates the root vertex v_{init} with the initial hybrid-system state and adds v_{init} to $\mathcal{T}$, i.e., $v_{\text{init}}.s = \mathcal{H}.s_{\text{init}}$, $\mathcal{T}.V = \{v_{\text{init}}\}$, and $\mathcal{T}.E = \emptyset$. The automaton states are computed by running $\mathcal{A}$ on the propositional assignment satisfied by $v_{\text{init}}.s$, i.e., $v_{\text{init}}.\alpha = \mathcal{A}.\delta(\mathcal{A}.z_{\text{init}}, \tau(v_{\text{init}}.s))$.

  (b) SelectVertexFromTree($\mathcal{P}, \mathcal{T}$) selects a vertex $v \in \mathcal{T}.V$ from which to extend $\mathcal{T}$. Over the years, numerous strategies have been proposed that rely on distances, nearest neighbors, probability distributions, and much more [23, 24].

  (c) ExtendTree($\mathcal{P}, \mathcal{T}, v$) extends $\mathcal{T}$ from v by computing a trajectory $\zeta : \mathbb{R}^{>0} \rightarrow \mathcal{H}.S$ that starts at $v.s$ and satisfies the invariant. A common strategy is to apply some input $u \in \mathcal{H}.U$ to $v.s$ and follow the dynamics of $\mathcal{H}$ until the invariant is not satisfied or a maximum number of steps is exceeded [8, 9, 10, 11, 12, 23, 24]. The input u is generally selected pseudo-uniformly at random to allow subsequent calls to extend $\mathcal{T}$ along new directions. ExtendTree($\mathcal{P}, \mathcal{T}, v$) returns a tuple $[u, T, s_{\text{new}}, \alpha_{\text{new}}]$, which defines $\zeta = \Upsilon_{v.s, u, T}$ (Section 2), where $s_{\text{new}} = \Upsilon_{v.s, u, T}(T)$ and $\alpha_{\text{new}} = \mathcal{A}(\tau(\text{TRAJ}(\mathcal{T}, v) \circ \zeta))$. Note that any hybrid-system

simulation method can be used to compute $\zeta = \Upsilon_{v.s,u,T}$. For completeness, we describe a simple iterative procedure. Let n_{steps} denote the number of steps and let $\epsilon > 0$ denote the step size (Algo. 3.1(c):1). Initially, $x_0 = x$ and $\alpha_0 = v.\alpha$, where $v.s = (q, x)$ (Algo. 3.1(c):2). At the i-th iteration, $x_i = \mathcal{H}.\text{FLOW}_q(x_{i-1}, u, \epsilon)$ (Algo. 3.1(c):5). The automaton states α_i associated with (q, x_i) are updated only if $\tau((q, x_i)) \neq \tau((q, x_{i-1}))$. The update is computed by running $\mathcal{A}$ on $\tau((q, x_i))$ starting from α_{i-1} (Algo. 3.1(c):6). If $\mathcal{H}.\text{INV}_q(x_i) = \bot$, then EXTENDTREE returns $[u, (i-1) * \epsilon, (q, x_{i-1}), \alpha_{i-1}]$ (Algo. 3.1(c):7). When $\mathcal{H}.\text{INV}_q(x_i) = \top$, EXTENDTREE checks if a guard is satisfied, which would indicate a discrete event (Algo. 3.1(c):8). Event detection is followed by event localization, which localizes the earliest time $T \in ((i-1) * \epsilon, i * \epsilon]$ where the guard is satisfied (Algo. 3.1(c):9). Bisection or bracketing algorithms are typically used for event localization [25]. The discrete transition is then triggered to obtain the new state (Algo. 3.1(c):11). The automaton states are also updated (Algo. 3.1(c):12).

Numerical errors in simulation, invariant checking, event detection and localization could in certain cases cause EXTENDTREE to miss an invariant violation, miss a guard, or trigger a different discrete transition. To minimize such errors, a practical approach is to choose a small ϵ. This approach is the norm in hybrid-system falsification methods based on motion planning [8, 9, 10, 11, 12]. For hybrid systems with linear guards, it is also possible to use more accurate event detection and localization algorithms, which come asymptotically close to the guard boundary[25]. In many practical cases, hybrid systems exhibit a degree of robustness [19, 26] that minimizes the impact of numerical errors, e.g., small perturbations do not change the mode-switching behavior. As noted, the simple implementation of EXTENDTREE, presented here for completeness, can be replaced by more sophisticated hybrid-system simulation methods.

(d) **AddBranchToTree**$(\mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}])$ adds v_{new} and (v, v_{new}) to $\mathcal{T}$. It also associates s_{new} and α_{new} with v_{new} and u and T with (v, v_{new}).

(e) **Traj**$(\mathcal{T}, v_{\text{new}})$ computes the trajectory from $v_{\text{init}}.s$ to $v_{\text{new}}.s$ by concatenating the trajectories associated with the tree edges connecting v_{init} to v_{new}.

Incorporating LTL into `RRT`: The work in [8, 9, 10] relies on `RRT`[14]. To incorporate LTL into `RRT`, it suffices to use `LTL-TSF` (Algo. 3.1) and implement SELECTVERTEXFROMTREE$(\mathcal{P}, \mathcal{T})$ as described in [8, 9, 10, 14], e.g., sample $s \in \mathcal{H}.S$ pseudo-uniformly at random and select $v \in \mathcal{T}.V$ whose $v.s$ is the closest to s according to a distance metric. This is referred to as `RRT[LTL-TSF]`.

Incorporating LTL into `HyDICE[NoGuide]`: Similarly to `RRT`, `HyDICE`[11, 12] also falls into the broad category of tree-search algorithms. Distinctly from `RRT`, `HyDICE` [11, 12] introduced discrete search over $(\mathcal{H}.Q, \mathcal{H}.E)$ to guide the tree search in the context of reachability analysis to a set of unsafe states. At each iteration, the discrete search computed a sequence of discrete transitions from an initial to an unsafe mode. The tree-search framework then extended $\mathcal{T}$ along the direction provided by the discrete search. Experiments showed significant speedup of one to two orders of magnitude over `RRT`-based falsification [9, 10].

Incorporating LTL into `HyDICE` is more involved than in the case of `RRT`, since the discrete search over $(\mathcal{H}.Q, \mathcal{H}.E)$ does not take LTL into account. When

considering LTL, a safety violation is not indicated by an unsafe state, but by an unsafe trajectory that satisfies $\neg\phi$. Therefore, when considering LTL, unsafe states and unsafe modes are not defined. This means that the discrete search over $(\mathcal{H}.Q, \mathcal{H}.E)$ from an initial to an unsafe mode is also not defined. The next section shows how to effectively incorporate LTL into HyDICE.

The version of HyDICE [11, 12] that does not use the discrete search is referred to in [11, 12] as HyDICE[NoGuide]. Experiments in [11, 12] showed that HyDICE[NoGuide] was significantly slower than HyDICE, but still faster than RRT-based falsification [9, 10]. As described in [11, 12], HyDICE[NoGuide] corresponds to the tree-search framework, where SELECTVERTEXFROMTREE$(\mathcal{P}, \mathcal{T})$ is implemented by selecting $v \in \mathcal{T}.V$ according to a probability distribution over $\mathcal{T}.V$. This makes it possible to incorporate LTL into HyDICE[NoGuide], referred to as HyDICE[NoGuide, LTL-TSF], by using LTL-TSF (Algo 3.1).

4   TemporalHyDICE

The computational efficiency of LTL-TSF (Algo. 3.1) depends on the ability of the approach to quickly extend $\mathcal{T}$ along those directions that lead to the computation of witness trajectories. Motivated by [11, 12], TemporalHyDICE uses a discrete transition model $\mathcal{M}$ of $\mathcal{H}$ and effectively combines LTL-TSF with model checking over $\mathcal{M}$ and $\mathcal{A}$ to identify and extend $\mathcal{T}$ along such useful directions.

Consider a *discrete witness* $[\tau_i]_{i=1}^n$, i.e., a sequence of propositional assignments accepted by $\mathcal{A}$. Let $\Gamma(\tau_i) = \{s \in \mathcal{H}.S : \tau(s) = \tau_i\}$. If $\mathcal{T}$ can be extended so that a trajectory TRAJ$(\mathcal{T}, v)$ starts at $\Gamma(\tau_1)$ and enters $\Gamma(\tau_2), \ldots, \Gamma(\tau_n)$ in succession, then TRAJ$(\mathcal{T}, v)$ would be a witness trajectory. In this way, the discrete witness provides a *feasible* direction along which motion planning in TemporalHyDICE can attempt to extend $\mathcal{T}$ in the search for a witness trajectory.

Model checking can be effectively employed for the computation of discrete witnesses. A discrete transition model is constructed as a graph $\mathcal{M} = (V, E)$ in order to capture the partition of $\mathcal{H}.S$ induced by τ, where a vertex $v(\tau_i) \in \mathcal{M}.V$ corresponds to $\Gamma(\tau_i)$ and an edge $(v(\tau_i), v(\tau_j)) \in \mathcal{M}.E$ indicates that it may be possible to enter directly from $\Gamma(\tau_i)$ to $\Gamma(\tau_j)$. Model checking can then compute discrete witnesses by simultaneously searching $\mathcal{A}$ and $\mathcal{M}$.

An issue that arises is which discrete witnesses motion planning can actually follow. Since it is not known a priori which discrete witnesses are feasible, TemporalHyDICE maintains a running weight estimate $w([\tau_i]_{i=1}^n)$ on the feasibility of $[\tau_i]_{i=1}^n$. A high weight indicates significant progress is made in extending $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$, while a low weight indicates little or no progress.

The core loop consists of using model checking to select at each iteration a discrete witness $[\tau_i]_{i=1}^n$ based on $w([\tau_i]_{i=1}^n)$ and then using motion planning to extend $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ in succession.

Combining Model Checking and Motion Planning: A crucial property of TemporalHyDICE, distinctive from earlier work [17], is that model checking and motion planning work in tandem. Information gathered by motion planning (such as coverage, $\Gamma(\tau_i)$'s that have been reached, and time spent) is used to update

---

Algorithm 4.1. TemporalHyDICE

Input: $\mathcal{P}$: problem specification; $t_{\max} \in \mathbb{R}^{>0}$: upper bound on computation time
Output: A witness trajectory if one is found or $\bot$ otherwise

---

(a) $\mathcal{T} \leftarrow$ INITIALIZETREE($\mathcal{P}$)
(b) $\mathcal{M} = (V, E) \leftarrow$ DISCRETETRANSITIONMODEL($\mathcal{P}$)
(c) INITIALIZEFEASIBILITYESTIMATE($\mathcal{P}, \mathcal{M}, w$)
while ELAPSEDTIME $< t_{\max}$ **do**
  (d) $\sigma \overset{def}{=} [(z_i, \tau_i)]_{i=1}^n \leftarrow$ DISCRETEWITNESS($\mathcal{P}, \mathcal{M}, w$)
  (e) $\zeta \leftarrow$ EXTENDTREEALONGDISCRETEWITNESS($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, \sigma$)
  (f) **if** $\zeta \neq$ NIL **return** ζ
return $\bot$

(e) EXTENDTREEALONGDISCRETEWITNESS($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, \sigma$) :=
1: $\sigma_{\mathrm{avail}} \leftarrow \{(z_i, \tau_i) \in \sigma : (z_i, \tau_i).\mathrm{vertices} \neq \emptyset\}$
2: **for** several times **do**
3:   $(z_i, \tau_i) \leftarrow$ SELECTAVAILABLEPAIR($w, \sigma_{\mathrm{avail}}$)
4:   $v \leftarrow$ SELECTVERTEXFROMAVAILABLEPAIR($w, (z_i, \tau_i).\mathrm{vertices}$)
5:   $[u, T, s_{\mathrm{new}}, \alpha_{\mathrm{new}}] \leftarrow$ EXTENDTREE($\mathcal{P}, \mathcal{T}, v$)
6:   **if** $T > 0 \wedge |\alpha_{\mathrm{new}}| > 0$ **then** $v_{\mathrm{new}} \leftarrow$ ADDBRANCHTOTREE($\mathcal{T}, v, [u, T, s_{\mathrm{new}}, \alpha_{\mathrm{new}}]$)
7:   **if** $\mathcal{P}.\mathcal{A}.\mathrm{Acc} \cap \alpha_{\mathrm{new}} \neq \emptyset$ **then return** TRAJ($\mathcal{T}, v_{\mathrm{new}}$)
8:   UPDATEFEASIBILITYESTIMATES($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, (z_i, \tau_i)$)
9:   $\tau_{\mathrm{new}} \leftarrow \mathcal{P}.\tau(v_{\mathrm{new}}.s)$
10:   **for** $z_{\mathrm{new}} \in \alpha_{\mathrm{new}}$ **do**
11:     $\sigma_{\mathrm{avail}} \leftarrow \{(z_{\mathrm{new}}, \tau_{\mathrm{new}})\} \cup \sigma_{\mathrm{avail}}$
12:     $(z_{\mathrm{new}}, \tau_{\mathrm{new}}).\mathrm{vertices} \leftarrow \{v_{\mathrm{new}}\} \cup (z_{\mathrm{new}}, \tau_{\mathrm{new}}).\mathrm{vertices}$
13:     UPDATEFEASIBILITYESTIMATES($\mathcal{P}, \mathcal{T}, \mathcal{M}, w, (z_{\mathrm{new}}, \tau_{\mathrm{new}})$)
14: **return** NIL

---

the feasibility estimates $w([\tau_i]_{i=1}^n)$. As a result, a new discrete witness, associated with a high weight, could be selected in the next iteration by model checking. In turn, by using highly feasible discrete witnesses $[\tau_i]_{i=1}^n$ as guides, motion planning is able to make progress and extend $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ until it successfully computes a witness trajectory. Pseudocode is given in Algo. 4.1.

Algo. 4.1(b) DiscreteTransitionModel($\mathcal{P}$): As discussed, $\mathcal{M}$ captures the partition of $\mathcal{H}.S$ induced by τ and serves to eliminate from consideration certain infeasible discrete witnesses. Region $\Gamma(\tau_j)$ is considered unable to directly reach $\Gamma(\tau_k)$, written $\Gamma(\tau_j) \not\rightarrow \Gamma(\tau_k)$, if $\Gamma(\tau_j)$ and $\Gamma(\tau_k)$ do not share a boundary and there is no discrete transition from some $s' \in \Gamma(\tau_j)$ to some $s'' \in \Gamma(\tau_k)$. A discrete witness $[\tau_i]_{i=1}^n$ is indeed infeasible if $\Gamma(\tau_k) \not\rightarrow \Gamma(\tau_{k+1})$ for some $1 \leq k < n$, since no trajectory can enter $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ in succession. To eliminate such infeasible discrete witnesses from consideration, $\mathcal{M}$ is constructed as a graph $\mathcal{M} = (V, E)$. A vertex $v(\tau_i)$ is added to $\mathcal{M}.V$ for each $\Gamma(\tau_i)$. An edge $(v(\tau_i), v(\tau_j))$ is added to $\mathcal{M}.E$ if it cannot be determined that $\Gamma(\tau_i) \not\rightarrow \Gamma(\tau_j)$.

Note that the computation of $\mathcal{M}$ is problem specific and depends on the black-box definitions of propositional, guards, and reset functions (Section 2). For this reason, DISCRETETRANSITIONMODEL($\mathcal{P}$) is an external function supplied by the user. Since there is no requirement that $\mathcal{M}$ should simulate $\mathcal{H}$, it is

generally a straightforward process for the user to obtain $\mathcal{M}$ from $\mathcal{P}$. This is the case for the experiments in this work. Moreover, the definition of $\mathcal{M}$ allows for spurious edges, i.e., $(v(\tau_j), v(\tau_k)) \in \mathcal{M}.E$ even when $\Gamma(\tau_j) \not\rightarrow \Gamma(\tau_k)$. This further facilitates the computation of $\mathcal{M}$ since the user can add spurious edges when it is computationally difficult to determine that $\Gamma(\tau_j) \not\rightarrow \Gamma(\tau_k)$. A spurious edge may cause model checking to compute at some iterations infeasible discrete witnesses, since it is impossible to enter directly from $\Gamma(\tau_j)$ to $\Gamma(\tau_k)$. The interplay between model checking and motion planning will cause feasibility estimates associated with spurious edges to decrease rapidly, since motion planning will fail to extend $\mathcal{T}$ from $\Gamma(\tau_j)$ to $\Gamma(\tau_k)$. As a result, model checking will reduce the likelihood of including spurious edges in future computations of discrete witnesses.

Algo. 4.1(d) DiscreteWitness$(\mathcal{P}, \mathcal{M}, w)$ uses model checking to compute discrete witnesses by searching on-the-fly $\mathcal{A}$ and $\mathcal{M}$. The search produces a sequence $[(z_i, \tau_i)]_{i=1}^{n}$, where $(z_i, \tau_i) \in \mathcal{A}.Z \times 2^{\Pi}$ and $z_n \in \mathcal{A}.\text{Acc}$. A critical issue is which discrete witness to select from combinatorially many possibilities. To address this issue, `TemporalHyDICE` associates a running estimate $w(z_i, \tau_i)$ on the feasibility of including (z_i, τ_i) in the current discrete witness. Let $(z_i, \tau_i).\text{vertices} = \{v \in \mathcal{T}.V : z_i \in v.\alpha \wedge \tau_i = \tau(v.s)\}$, i.e., v is associated with (z_i, τ_i) iff $v.s$ satisfies τ_i and z_i is included in the automaton states $v.\alpha$ obtained by running $\tau(\text{TRAJ}(\mathcal{T}, v))$ on $\mathcal{A}$. Then,

$$w(z_i, \tau_i) = \text{cov}^{a_1}(z_i, \tau_i) * \text{vol}^{a_2}(\Gamma(\tau_i))/\text{time}(z_i, \tau_i), \tag{1}$$

where $\text{cov}(z_i, \tau_i)$ estimates the coverage of $\Gamma(\tau_i)$ by the states associated with $(z_i, \tau_i).\text{vertices}$; $\text{vol}(\Gamma(\tau_i))$ is the volume of $\Gamma(\tau_i)$; $\text{time}(z_i, \tau_i)$ is the time motion planning has spent extending $\mathcal{T}$ from $(z_i, \tau_i).\text{vertices}$; and a_1, a_2 are normalization constants. The combination of coverage, volume, and computational time is motivated by motion planners for continuous and hybrid systems [8, 9, 10, 11, 12, 27]. As in [11, 12], $\text{cov}(z_i, \tau_i)$ is computed by imposing an implicit uniform grid on a low-dimensional projection of $\mathcal{H}.S$ and counting the number of grid cells that have at least one state from the states associated with $(z_i, \tau_i).\text{vertices}$. The volume $\text{vol}(\Gamma(\tau_i))$ is a user-supplied value, since it depends on the black-box definitions of the proposition functions $\text{PROP}_{q,i}$ (Section 2). In the experiments in this work, $\text{PROP}_{q,i}$ define polygons and $\text{vol}(\Gamma(\tau_i))$ is computed as the corresponding polygonal area. `TemporalHyDICE` associates a high weight $w(z_i, \tau_i)$ with (z_i, τ_i) if motion planning has extended $\mathcal{T}$ toward a region $\Gamma(\tau_i)$ with a large volume, and states associated with $(z_i, \tau_i).\text{vertices}$ quickly cover $\Gamma(\tau_i)$.

The discrete witness is computed as the shortest path from initial to accepting states by using Dijkstra's algorithm, where an edge $((z_i, \tau_i), (z_j, \tau_j))$ is assigned the weight $1/(w(z_i, \tau_i) * w(z_j, \tau_j))$. This allows to select highly feasible discrete witnesses. With small probability, the discrete witness is also computed as a random path using a variation of the depth-first-search, where the frontier nodes are visited in a random order. This randomness provides a way to correct for errors inherent with the weight estimates by ensuring that each discrete witness that is not determined as infeasible is selected with non-zero probability.

`TemporalHyDICE` does not explicitly construct $\mathcal{A} \times \mathcal{M}$. During the search for a discrete witness, the outgoing edges of (z_i, τ_i) are computed implicitly

as $\text{EDGES}(z_i, \tau_i) = \{(z_j, \tau_j) : (v(\tau_i), v(\tau_j)) \in \mathcal{M}.E \wedge z_j \in \mathcal{A}.\delta(z_i, \tau_j)\}$. This allows $\texttt{TemporalHyDICE}$ to considerably reduce the memory requirements of model checking. Note that the largest memory requirements in $\mathcal{A}$ are imposed by $\mathcal{A}.\delta$, which can be viewed as a ternary relation, subset of $\mathcal{A}.Z \times \Sigma \times \mathcal{A}.Z$, where $\Sigma = 2^{\Pi}$. On the other hand, $\mathcal{M}$ can be viewed as a binary relation, subset of $\Sigma \times \Sigma$. Explicitly constructing $\mathcal{A} \times \mathcal{M}$ would produce a 4-ary relation, subset of $\mathcal{A}.Z \times \Sigma^2 \times \mathcal{A}.Z$. For this reason, $\texttt{TemporalHyDICE}$ does not compute $\mathcal{A} \times \mathcal{M}$ explicitly. In addition, the data structure that stores information about a pair (z_i, τ_i) is created only when a vertex v is added to $\mathcal{T}.V$ such that $z_i \in v.\alpha$ and $\tau_i = \tau(v.s)$. Reducing memory requirements is important for $\texttt{TemporalHyDICE}$, since it allows motion planning to extend $\mathcal{T}$ by adding more vertices and edges.

Algo. 4.1(e) ExtendTreeAlongDiscreteWitness$(\mathcal{P}, \mathcal{T}, \mathcal{M}, w, \sigma)$**:** Let $\sigma = [(z_i, \tau_i)]_{i=1}^n$ denote the current discrete witness. The objective is to extend $\mathcal{T}$ so that it reaches $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ in succession. To achieve this objective, the method proceeds by extending $\mathcal{T}$ from vertices associated with pairs (z_i, τ_i).

(Algo. 4.1(e):1) Only pairs $(z_i, \tau_i) \in \sigma$ reached by $\mathcal{T}$, i.e., $(z_i, \tau_i).\text{vertices} \neq \emptyset$, can be considered for selecting a vertex v from which to extend $\mathcal{T}$.

(Algo. 4.1(e):3) $\text{SELECTAVAILABLEPAIR}(w, \sigma_{\text{avail}})$ selects a pair (z_i, τ_i) from σ_{avail} with probability $w(z_i, \tau_i)/\sum_{(z_j, \tau_j) \in \sigma_{\text{avail}}} w(z_j, \tau_j)$, where $w(z_i, \tau_i)$ is defined in Eqn. 1. This selection, thus, favors highly feasible pairs.

(Algo. 4.1(e):4) $\text{SELECTVERTEXFROMAVAILABLEPAIR}(w, (z_i, \tau_i).\text{vertices})$ selects a vertex v from $(z_i, \tau_i).\text{vertices}$ with probability $\frac{1}{\text{nsel}(v)}/\sum_{v' \in (z_i, \tau_i).\text{vertices}} \frac{1}{\text{nsel}(v')}$, where $\text{nsel}(v)$ is one plus the number of times v has been selected in the past from $(z_i, \tau_i).\text{vertices}$. This is based on well-established strategies in motion planning that favor those vertices selected less frequently in the past [23, 24].

(Algo. 4.1(e):5–7) As described in Section 3, $\text{EXTENDTREE}(\mathcal{P}, \mathcal{T}, v)$ and $\text{ADDBRANCHTOTREE}(\mathcal{P}, \mathcal{T}, v, [u, T, s_{\text{new}}, \alpha_{\text{new}}])$ extend $\mathcal{T}$ from v by computing and adding to $\mathcal{T}$ a valid trajectory that starts at $v.s$. If any of the automaton states α_{new} is an accepting state, then $\text{TRAJ}(\mathcal{T}, v_{\text{new}})$ is a witness trajectory.

(Algo. 4.1(e):8–13) The feasibility estimate associated with (z_i, τ_i) is updated to reflect the extension of $\mathcal{T}$ from v. The vertex v_{new} is associated with each $(z_{\text{new}}, \tau_{\text{new}})$, where $z_{\text{new}} \in \alpha_{\text{new}}$ and $\tau_{\text{new}} = \tau(v_{\text{new}}.s)$. The feasibility estimate $w(z_{\text{new}}, \tau_{\text{new}})$ is also updated to reflect the addition of v_{new} to $(z_{\text{new}}, \tau_{\text{new}}).\text{vertices}$. Each $(z_{\text{new}}, \tau_{\text{new}})$ is also added to σ_{avail}, so that it becomes available for selection in the next iteration. The updated weights better estimate the feasibility of each discrete witness, and thus improve the selection of discrete witnesses for the next iteration. This in turn allows motion planning to make more progress in extending $\mathcal{T}$ toward $\Gamma(\tau_1), \ldots, \Gamma(\tau_n)$ and eventually compute a witness trajectory.

5   Experiments and Results

The experiments provide an initial validation of $\texttt{TemporalHyDICE}$ for the falsification of safety properties expressed by syntactically safe LTL formulas for hybrid systems with nonlinear dynamics. $\texttt{TemporalHyDICE}$ is shown to be significantly more efficient than the straightforward extensions of related work [8, 9, 10, 11, 12], which use the automaton $\mathcal{A}$ as an external monitor (see Section 3). The experiments also demonstrate the importance of model checking and the discrete

transition model in the computational efficiency of `TemporalHyDICE`. This paper also studies the impact of $\mathcal{A}$ (NFA or DFA) on the efficiency of `TemporalHyDICE`.

The hybrid system $\mathcal{H}$ models an autonomous vehicle driving over different terrains, similar to the navigation benchmark proposed in [28] and used in [11, 12]. Each terrain corresponds to a mode $q \in \mathcal{H}.Q$. The dynamics, velocity, and acceleration vary from one terrain to another. Second-order dynamics (with 5 dimensions) for modeling cars, differential drives, and unicycles (see [11, 23, 24] for model details) are associated with each mode. In each terrain, several polygons are marked as propositions $\text{PROP}_{q_i,k}$ and guards GUARD_{q_i,q_j}. A state $s = (q, x) \in \mathcal{H}.S$ satisfies $\text{PROP}_{q_i,k}$ (resp., GUARD_{q_i,q_j}) iff $q = q_i$ and the position-component of x is inside $\text{PROP}_{q_i,k}$ (resp., GUARD_{q_i,q_j}). When GUARD_{q_i,q_j} is satisfied, a discrete transition occurs. The mode is then set to q_j and velocity is set to zero.

The choice of this specific system is to provide a concrete benchmark that is easily scalable to test `TemporalHyDICE` as the complexity of LTL formulas is increased. For the experiments, 12 safety properties and 100 instances of the benchmark were created. Syntactically safe LTL formulas were manually designed in order to provide meaningful properties. Benchmark instances were generated at random in order to test `TemporalHyDICE` over many problems and obtain statistically significant results. Experimental data is publicly available.[1]

Problem Instances: In each problem instance, number of modes is $n_Q = 10$, number of propositions per mode is $n_P = 15$, and number of guards per mode is $n_G = 5$. A random problem instance is generated as follows. First, the second-order dynamics associated with each mode is selected pseudo-uniformly at random from those of a car, unicycle, or differential drive. Second, velocity is bounded by $v_{\max}$, where $v_{\max}$ is selected pseudo-uniformly at random from $[3, 6]$m/s. Third, for each mode, n_P propositions and n_G guards are generated as random polygons. Let $\pi_1, \ldots, \pi_{150}$ denote the generated propositions.

Syntactically-Safe LTL Formulas: Let $\beta_0 = \neg(\pi_1 \vee \cdots \vee \pi_{150})$.
– sequencing ($n = 3, 4, 5, 6$): Witness trajectory will reach $\pi_1, \ldots, \pi_n$ in order:
$\phi_1^n = \neg\left(\beta_0 \mathcal{U}\left(\pi_1 \wedge \left(\pi_1 \mathcal{U}\left(\pi_2 \wedge \left(\pi_2 \mathcal{U}\left(\ldots \pi_{n-1} \wedge \left(\pi_{n-1} \mathcal{U}\left(\beta_0 \mathcal{U} \pi_n\right)\right)\right)\right)\right)\right)\right)\right)$;
– counting ($n = 1, 2, 3, 4$): Witness trajectory will reach π_2, π_3, π_4 n-times in order, and then it will reach π_5: $\phi_2^n = \neg(\varsigma_1 \mathcal{U}(\pi_1 \wedge \Xi_1(\Xi_2 \cdots (\Xi_n(\varsigma_1 \mathcal{U} \pi_5)))))$,
$\Xi_j(\psi) \overset{def}{=} \varsigma_1 \mathcal{U}\left(\pi_2 \wedge \left(\varsigma_2 \mathcal{U}\left(\pi_3 \wedge \left(\varsigma_3 \mathcal{U}\left(\pi_4 \wedge \left(\pi_4 \mathcal{U}\left(\varsigma_1 \wedge \psi\right)\right)\right)\right)\right)\right)\right)$; $\varsigma_i \overset{def}{=} \beta_0 \vee \pi_i$;
– coverage ($n = 4, 5, 6, 7$): Witness trajectory reacheas each π_i: $\phi_3^n = \bigvee_{i=1}^n \mathcal{G}(\neg \pi_i)$.

Results: Experiments were run on Rice Cray XD1 ADA and PBC clusters. Each run uses a single processor (2.2Ghz, 8GB RAM), i.e., no parallelism. The automata for each $\neg\phi$ are computed by standard tools (scheck [29]). Comparisons of `TemporalHyDICE` to `RRT[LTL-TSF]` in Table 1(a) provide a basis for the results. While `TemporalHyDICE` solved all problem instances, `RRT[LTL-TSF]` timed out in almost every instance. `RRT[LTL-TSF]` relies on distance metrics and nearest neighbors to guide the search. By relying on such limited information, as shown in [11, 12] in the context of reachability analysis, it quickly becomes difficult to find feasible directions to extend $\mathcal{T}$, causing a rapid decline in the growth of $\mathcal{T}$. The results in Table 1(a) confirm this observation also in the case of applying

---

[1] http://www.kavrakilab.org/data/TACAS2009/

Table 1. Reported is the average time in seconds to solve 100 problem instances for each of the LTL formulas. Times for `TemporalHyDICE` include the construction of $\mathcal{M}$, which took < 1s. Entries marked with X indicate a timeout (set to 400s).

(a) Comparison of different methods.												
LTL safety formula	ϕ_1^3	ϕ_1^4	ϕ_1^5	ϕ_1^6	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4	ϕ_3^4	ϕ_3^5	ϕ_3^6	ϕ_3^7
nr. states minimized DFA	10	21	46	105	23	76	164	287	16	32	64	128
`TemporalHyDICE`	18.6	25.5	27.2	40.4	22.2	40.4	63.3	88.3	14.6	40.9	127.9	293.2
`RRT[LTL-TSF]`	267.2	X	X	X	X	X	X	X	X	X	X	X
`HyDICE[NoGuide,LTL-TSF]`	245.3	X	X	X	X	X	X	X	X	X	X	X
`TemporalHyDICE[no `$\mathcal{M}$`]`	19.2	55.7	X	X	203.8	X	X	X	76.2	367.5	X	X

(b) Comparison of `TemporalHyDICE` when using a minimal DFA, a minimal NFA constructed by hand, or an NFA constructed by standard tools for ϕ_2^n, $n = 1, 2, 3, 4$.												
	Minimized DFA				Minimized NFA				Standard NFA			
LTL safety formula	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4	ϕ_2^1	ϕ_2^2	ϕ_2^3	ϕ_2^4
nr. states in automaton	23	76	164	287	7	11	15	19	27	176	912	4099
`TemporalHyDICE`	22.2	40.4	63.3	88.3	23.5	37.6	52.5	74.4	86.2	X	X	X

`RRT[LTL-TSF]` to falsify LTL safety properties in hybrid systems. By combining model checking and motion planning, `TemporalHyDICE` effectively guides the tree search. We also observe that the running time of `TemporalHyDICE` increases sub-linearly (ϕ_1^n and ϕ_2^n) or sub-quadratically (ϕ_3^n) with the number of states in the minimized DFA. These results provide promising initial validation.

Comparisons of `TemporalHyDICE` to `HyDICE[NoGuide,LTL-TSF]` in Table 1(a) demonstrate the importance of combining model checking and motion planning. Without model checking to guide motion planning, `HyDICE[NoGuide,LTL-TSF]`, similar to `RRT[LTL-TSF]`, times out in almost all instances.

Comparisons of `TemporalHyDICE` to `TemporalHyDICE[no `$\mathcal{M}$`]` in Table 1(a) indicate the importance of computing discrete witnesses by searching $\mathcal{M}$ and $\mathcal{A}$ (as in `TemporalHyDICE`) and not just $\mathcal{A}$ (as in `TemporalHyDICE[no `$\mathcal{M}$`]`). When searching just $\mathcal{A}$, a discrete witness may contain propositional assignments τ_i and τ_{i+1} that cannot be satisfied consecutively, i.e., $\Gamma(\tau_i) \not\rightarrow \Gamma(\tau_{i+1})$. As discussed in Section 4, $\mathcal{M}$ serves to eliminate from consideration many of these infeasible discrete witnesses. This in turn speeds up the search for a witness trajectory since $\mathcal{T}$ is extended far more frequently toward feasible directions. It is also important to note that, even though the discrete witnesses obtained by searching just $\mathcal{A}$ are not as beneficial as those obtained by searching $\mathcal{M}$ and $\mathcal{A}$, `TemporalHyDICE[no `$\mathcal{M}$`]` is still considerably faster than methods that do not guide the tree search, cf. `RRT[LTL-TSF]` and `HyDICE[NoGuide,LTL-TSF]`.

Table 1(b) compares `TemporalHyDICE` when using NFAs computed by standard tools (scheck [29]), minimal NFAs constructed by hand, or minimal DFAs. These experiments are motivated by the work in [21], which shows significant speedup when using DFAs instead of NFAs in model checking. As Table 1(b) shows, `TemporalHyDICE` is only slightly faster when using minimal NFAs instead of minimal DFAs, even though minimal NFAs had significantly fewer states. As concluded in [21], DFA search has a significantly smaller branching factor than NFA search, which allows it to offset the drawbacks of a possibly exponential increase in the size of DFA. This observation is also supported by comparisons of

minimal DFAs to standard NFAs, since in such cases there is significant speedup when using minimal DFAs. Therefore, the non-minimized NFA should be determinized and minimized.

6   Discussion

This work developed a novel method, `TemporalHyDICE`, for the falsification of safety properties specified by syntactically safe LTL formulas for hybrid systems with general nonlinear dynamics. By effectively combining model checking and motion planning, when a hybrid system is unsafe, `TemporalHyDICE` may compute a witness trajectory that indicates a violation of the safety property. Experiments show significant speedup over related work. As we consider more complex safety properties and high-dimensional continuous systems, it becomes important to further improve the synergistic combination of model checking and motion planning. Another direction is to extend the theory developed in [30] to show probabilistic completeness for `TemporalHyDICE`.

Acknowledgment

References

1. Tomlin, C.J., Mitchell, I., Bayen, A., Oishi, M.: Computational techniques for the verification and control of hybrid systems. Proc. of IEEE 91(7), 986–1001 (2003)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
3. Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: ACM Symp. on Theory of Computing, pp. 373–382 (1995)
4. Mitchell, I.M.: Comparing forward and backward reachability as tools for safety analysis. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 428–443. Springer, Heidelberg (2007)
5. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.: Discrete abstractions of hybrid systems. Proc. of IEEE 88(7), 971–984 (2000)
6. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and Counterexample-guided Refinement in Model Checking of Hybrid Systems. Intl. J. of Foundations of Computer Science 14(4), 583–604 (2003)
7. Giorgetti, N., Pappas, G.J., Bemporad, A.: Bounded model checking for hybrid dynamical systems. In: Conf. on Decision & Control, Seville, Spain, pp. 672–677 (2005)
8. Bhatia, A., Frazzoli, E.: Incremental search methods for reachability analysis of continuous and hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 142–156. Springer, Heidelberg (2004)
9. Kim, J., Esposito, J.M., Kumar, V.: An RRT-based algorithm for testing and validating multi-robot controllers. In: Robotics: Science & Systems, Boston, MA, pp. 249–256 (2005)
10. Nahhal, T., Dang, T.: Test coverage for continuous and hybrid systems. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 449–462. Springer, Heidelberg (2007)

11. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: From verification to falsification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 463–476. Springer, Heidelberg (2007)
12. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: From verification to falsification by combining motion planning and discrete search. Formal Methods in System Design (2008)
13. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 436–453. Springer, Heidelberg (2001)
14. LaValle, S.M., Kuffner, J.J.: Randomized kinodynamic planning. Intl. J. of Robotics Research 20(5), 378–400 (2001)
15. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
16. Behrmann, G., David, A., Larsen, K.G., Möller, O., Pettersson, P., Yi, W.: Uppaal present and future. In: Conf. on Decision & Control, Orlando, FL, pp. 2881–2886 (2001)
17. Fainekos, G.E., Kress-Gazit, H., Pappas, G.: Temporal logic motion planning for mobile robots. In: IEEE Intl. Conf. on Robotics & Automation, Barcelona, Spain, pp. 2020–2025 (2005)
18. Batt, G., Belta, C., Weiss, R.: Temporal logic analysis of gene networks under parameter uncertainty. IEEE Trans. of Automatic Control 53, 215–229 (2008)
19. Damm, W., Pinto, G., Ratschan, S.: Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. Intl. J. of Foundations of Computer Science 18(1), 63–86 (2007)
20. Kupferman, O., Vardi, M.: Model checking of safety properties. Formal methods in System Design 19(3), 291–314 (2001)
21. Armoni, R., Egorov, S., Fraer, R., Korchemny, D., Vardi, M.: Efficient LTL compilation for SAT-based model checking. In: Intl. Conf. on Computer-Aided Design, San Jose, CA, pp. 877–884 (2005)
22. Sistla, A.: Safety, liveness and fairness in temporal logic. Formal Aspects of Computing 6, 495–511 (1994)
23. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Cambridge (2005)
24. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
25. Esposito, J., Kumar, V., Pappas, G.: Accurate event detection for simulation of hybrid systems. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 204–217. Springer, Heidelberg (2001)
26. Julius, A.A., Fainekos, G.E., Anand, M., Lee, I., Pappas, G.J.: Robust test generation and coverage for hybrid systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 329–342. Springer, Heidelberg (2007)
27. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Discrete search leading continuous exploration for kinodynamic motion planning. In: Robotics: Science & Systems, Atlanta, GA (2007)
28. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)
29. Latvala, T.: Efficient model checking of safety properties. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 74–88. Springer, Heidelberg (2003)
30. Ladd, A.M.: Motion Planning for Physical Simulation. PhD thesis, Rice University, Houston, TX (2006)

Computing Optimized Representations for Non-convex Polyhedra by Detection and Removal of Redundant Linear Constraints

Christoph Scholl, Stefan Disch, Florian Pigorsch, and Stefan Kupferschmid

Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg, Germany

Abstract. We present a method which computes optimized representations for non-convex polyhedra. Our method detects so-called redundant linear constraints in these representations by using an incremental SMT (Satisfiability Modulo Theories) solver and then removes the redundant constraints based on Craig interpolation. The approach is motivated by applications in the context of model checking for Linear Hybrid Automata. Basically, it can be seen as an optimization method for formulas including arbitrary boolean combinations of linear constraints and boolean variables. We show that our method provides an essential step making quantifier elimination for linear arithmetic much more efficient. Experimental results clearly show the advantages of our approach in comparison to state-of-the-art solvers.

1 Introduction

In this paper we present an approach which uses SMT (Satisfiability Modulo Theories) solvers and Craig interpolation [1] for optimizing representations of non-convex polyhedra. Non-convex polyhedra are formed by arbitrary boolean combinations (including conjunction, disjunction and negation) of linear constraints. Non-convex polyhedra have been used to represent sets of states of hybrid systems. Whereas approaches like [2,3] consider unions of convex polyhedra (i.e. unions of conjunctions of linear constraints) together with an explicit representation of discrete states, in [4,5] a data structure called LinAIGs was used as a single symbolic representation for sets of states of hybrid systems with large discrete state spaces (in the context of model checking by backward analysis). LinAIGs in turn represent an extension of non-convex polyhedra by additional boolean variables, i.e., they represent arbitrary boolean combinations of boolean variables and linear constraints.

In particular, our optimization methods for non-convex polyhedra remove so-called *redundant linear constraints* from our representations. A linear constraint is called redundant for a non-convex polyhedron if and only if the non-convex polyhedron can be described without using this linear constraint. Note that an alternative representation of the polyhedron without using the redundant linear constraint may require a completely different boolean combination of linear

constraints. In that sense our method significantly extends results for eliminating redundant linear constraints from convex polyhedra used by Frehse [3] and Wang [6].[1] In previous work [5] we already made the observation that a major obstacle to the application of sequences of quantifier eliminations in the context of model checking for hybrid systems is formed by the growth of the number of linear constraints in state set representations during Weispfennig–Loos quantifier elimination [7]. For that reason removing redundant linear constraints from non-convex polyhedra plays an essential role during model checking of non-trivial examples.

Our paper makes the following contributions:

- We present an algorithm for detecting a maximal number of linear constraints which can be removed *simultaneously*. The algorithm is based on sets of don't cares which result from inconsistent assignments of truth values to linear constraints. We show how the *detection* of sets of redundant constraints can be performed using an SMT solver. In particular we show how to use *incremental* SMT solving for detecting larger and larger sets of redundant constraints until a maximal set is obtained.
- We show how the information needed for *removing* redundant linear constraints can be extracted from the conflict clauses of an SMT solver. Finally, we present a novel method really performing the removal of redundant linear constraints based on this information. The method is based on *Craig interpolation* [1,8,9].

In a comparison with existing tools we consider formulas consisting of arbitrary boolean combinations of linear constraints and boolean variables, combined with quantifications of real-valued variables. For such formulas we solve two problems: First, we compute whether the resulting formula is satisfiable by any assignment of values to the free variables and secondly we do even more, we also compute a predicate over the free variables which is true for all satisfying assignments of the formula. We compare our results to the results of the automata-based tool LIRA [10], the computer algebra system REDUCE/REDLOG [11,12] (which also solve both problems mentioned above) and to the results of state-of-the-art SMT solvers Yices [13] and CVC3 [14] (which solve the first problem of checking whether the formula is satisfiable). Whereas these solvers are not restricted to the subclass of formulas we consider in this paper (and are not optimized for this subclass in the case of Yices and CVC3), our experiments show that for the subclass of formulas considered here our method is much more effective. Our results are obtained by an elaborate scheme combining several methods for keeping representations of intermediate results compact with redundancy removal as an essential component. Internally, these methods make heavy use of the results of SMT solvers restricted to quantifier-free satisfiability solving.[2] Our

---

[1] For convex polyhedra redundancy of linear constraints reduces to the question whether the linear constraint can be omitted in the conjunction of linear constraints without changing the represented set.

[2] In our implementation we use Yices [13] and MathSAT [15] for this task.

results suggest to make use of our approach, if the formula at hand belongs to the subclass of linear arithmetic with quantification over reals and moreover, even for more general formulas, one can imagine to use our method as a fast preprocessor for simplifying subformulas from this subclass.

The paper is organized as follows: In Sect. 2 we give a brief review of our representations of non-convex polyhedra and Weispfennig-Loos quantifier elimination. In Sect. 3 we give a definition of redundant linear constraints and present methods for detecting and removing them from representations of non-convex polyhedra. After presenting our encouraging experimental results in Sect. 4 we conclude the paper in Sect. 5.

2 Preliminaries

2.1 Representation of Non-convex Polyhedra

We assume disjoint sets of variables C and B. The elements of $C = \{c_1, \ldots, c_f\}$ are continuous variables, which are interpreted over the reals $\mathbb{R}$. The elements of $B = \{b_1, \ldots, b_k\}$ are boolean variables and range over the domain $\mathbb{B} = \{0, 1\}$. When we consider logic formulas over $B \cup C$, we restrict terms over C to the class of linear terms of the form $\sum \alpha_i c_i + \alpha_0$ with rational constants α_i and $c_i \in C$. Predicates are based on the set $\mathcal{L}(C)$ of linear constraints, they have the form $t \sim 0$, where $\sim \in \{=, <, \leq\}$ and t is a linear term. $\mathcal{P}(C)$ is the set of all boolean combinations of linear constraints over C, the formulas from $\mathcal{P}(C)$ represent *non-convex polyhedra* over $\mathbb{R}^f$. In this paper we consider the class of formulas from $\mathcal{P}(B, C)$ which is the set of all boolean combinations of boolean variables from B and linear constraints over C.

As a underlying data structure for our method we use representations of formulas from $\mathcal{P}(B, C)$ by LinAIGs [4,5]. LinAIGs are And-Inverter-Graphs (AIGs) enriched by linear constraints. The structure of LinAIGs is illustrated in Fig. 1.
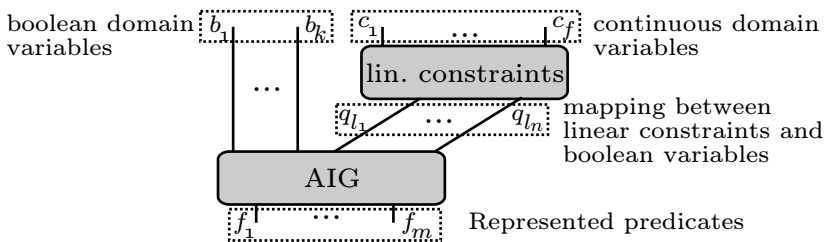
Fig. 1. Structure of LinAIGs

The component of LinAIGs representing boolean formulas consists in a variant of AIGs, the so–called Functionally Reduced AND-Inverter Graphs (FRAIGs) [16,17]. AIGs enjoy a widespread application in combinational equivalence checking and Bounded Model Checking (BMC). They are basically boolean circuits consisting only of AND gates and inverters. In [17] FRAIGs were tailored towards

the representation and manipulation of sets of states in symbolic model checking, replacing BDDs as a compact representation of large discrete state spaces.

In LinAIGs (see Fig. 1) we use a set of new (boolean) *constraint variables* Q as encodings for the linear constraints, where each occurring $\ell_i \in \mathcal{L}(C)$ is encoded by some $q_{\ell_i} \in Q$. For keeping the representation as compact as possible we use a multitude of methods; e.g., inserting different nodes representing the same predicate is avoided using an SMT (SAT modulo theories) solver which combines DPLL with linear programming as a decision procedure [4,5].

2.2   Quantifier Elimination

In [5] Loos's and Weispfenning's test point method [7] was adapted to the LinAIG data structure described above. The method eliminates universal quantifiers for real-valued variables by converting them into finite conjunctions and existential quantifiers by converting them into finite disjunctions. The subformulas to be combined by conjunction (or disjunction in case of existential quantification) are obtained from the original formula by replacing real-valued variables by appropriate 'test points' arriving again at formulas in linear arithmetic. The test point method is well-suited for our LinAIG data structure, since substitutions and disjunctions / conjunctions can be performed efficiently in the LinAIG package and the method does not need (potentially costly) conversions of the original formula into CNF / DNF before applying quantifier elimination as in the Fourier-Motzkin algorithm, e.g..

The number of test points needed depends linearly on the number of linear constraints in the original formula. Thus, during elimination of one real-valued variable, the number of linear constraints may grow quadratically with the number of linear constraints in the original formula. For sequences of quantifier eliminations it is therefore important to keep the number of linear constraints in the original formula as small as possible. Moreover, after elimination of one quantifier (starting with the innermost), it is also important to remove redundant linear constraints generated as a result of the test point method. For this reason we developed an algorithm which computes representations depending on a minimal set of linear constraints (see Sect. 3). Experimental results in Sect. 4 show that the method is indeed essential in order to enable sequences of quantifier eliminations.

3   Redundant Linear Constraints

In this section we present our methods to detect and remove redundant linear constraints from non-convex polyhedra. Note that our approach also *works for the generalized case of arbitrary boolean combinations of linear constraints and additional boolean variables* (as represented by LinAIGs, e.g.), but here we confine ourselves to non-convex polyhedra in order to keep the exposition more compact and readable.

For illustration of redundant linear constraints see Fig. 2 and 3, which show a typical example stemming from a model checking application. It represents a

small state set based on two real variables: Lines in Figures 2 and 3 represent linear constraints, and the gray shaded area represents the space defined by some boolean combination of these constraints. Whereas the representation depicted in Fig. 2 contains 24 linear constraints, a closer analysis shows that an optimized representation can be found using only 15 linear constraints as depicted in Fig. 3.
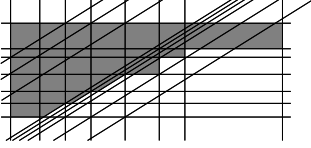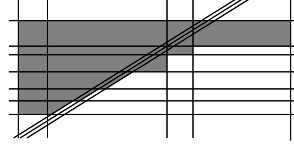

Fig. 2. Before redundancy removal

Fig. 3. After redundancy removal

3.1 Redundancy Detection and Removal for Convex Polyhedra

The task of detecting and removing redundant constraints in *non-convex* polyhedra is not as straightforward as for other approaches such as [2,3] which represent sets of *convex* polyhedra, i. e., sets of conjunctions $\ell_1 \wedge \ldots \wedge \ell_n$ of linear constraints. If one is restricted to convex polyhedra, the question whether a linear constraint ℓ_1 is redundant in the representation reduces to the question whether $\ell_2 \wedge \ldots \wedge \ell_n$ represents the same polyhedron as $\ell_1 \wedge \ldots \wedge \ell_n$, or equivalently, whether $\overline{\ell_1} \wedge \ell_2 \wedge \ldots \wedge \ell_n$ represents the empty set. This question can simply be answered by a linear program solver.

3.2 Detection of Redundant Constraints for Non-convex Polyhedra

Definition 1 (Redundancy of linear constraints). *Let F be a boolean function and let $\ell_1, \ldots, \ell_n$ be linear constraints over real-valued variables $C = \{c_1, \ldots, c_f\}$. The linear constraints $\ell_1, \ldots, \ell_r$ ($1 \leq r \leq n$) are called redundant in the representation of $F(\ell_1, \ldots, \ell_n)$ iff there is a boolean function G with the property that $F(\ell_1, \ldots, \ell_n)$ and $G(\ell_{r+1}, \ldots, \ell_n)$ represent the same predicates.*

Our check for redundancy is based on the following theorem [5]:

Theorem 1 (Redundancy check). *For all $1 \leq i \leq n$ let ℓ_i be a linear constraint over real-valued variables $\{c_1, \ldots, c_f\}$ and ℓ'_i **exactly the same** linear constraint as ℓ_i, but now over a **disjoint copy** $\{c'_1, \ldots, c'_f\}$ of the variables. Let $\oplus$ denote exclusive-or and $\equiv$ denote boolean equivalence. The linear constraints $\ell_1, \ldots, \ell_r$ ($1 \leq r \leq n$) are redundant in the representation of $F(\ell_1, \ldots, \ell_n)$ if and only if the predicate*

$$(F(\ell_1, \ldots, \ell_n) \oplus F(\ell'_1, \ldots, \ell'_n)) \wedge \bigwedge_{i=r+1}^{n}(\ell_i \equiv \ell'_i) \tag{1}$$

is not satisfiable by any assignment of real values to the variables $c_1, \ldots, c_f$ and $c'_1, \ldots, c'_f$.

Note that the check from Thm. 1 can be performed by a (conventional) SMT solver.

A sketch of the proof for the 'only-if-part' of Thm. 1 was already given in [5]. In this paper (Sect. 3.3) we present a constructive proof for the 'if-part' of the theorem in order to provide an efficient procedure to compute an appropriate function G whenever formula (1) is unsatisfiable.

Overall algorithm for redundancy detection. First of all, we present our overall algorithm detecting a maximal set of linear constraints which can be removed from the representation *at the same time*. We start with a small example demonstrating the effect that it is not enough to consider redundancy of single linear constraints and to construct larger sets of redundant constraints simply as unions of smaller sets.

Example 1. Consider the predicate $F(c_1, c_2) = (c_1 \geq 0) \wedge (c_2 \geq 0) \wedge \neg(c_1 + c_2 \leq 0) \wedge \neg(2c_1 + c_2 \leq 0)$. It is easy to see that both the third and the forth linear constraint in the conjunction have the effect of 'removing the value $(c_1, c_2) = (0, 0)$ from the predicate $F'(c_1, c_2) = (c_1 \geq 0) \wedge (c_2 \geq 0)$'. Therefore both $\ell_3 = (c_1 + c_2 \leq 0)$ and $\ell_4 = (2c_1 + c_2 \leq 0)$ are obviously redundant linear constraints in F. However, it is also easy to see that ℓ_3 and ℓ_4 are not redundant in the representation of F *at the same time*, i.e., only $\neg(c_1 + c_2 \leq 0)$ *or* $\neg(2c_1 + c_2 \leq 0)$ can be omitted in the representation for F.

This observation motivates the following overall algorithm to detect a maximal set of redundant linear constraints:

Input   : Predicate $F(\ell_1, \ldots, \ell_n)$
Output: S: Maximal set of redundant linear constraints
begin
  $S := \emptyset$;
  for $i := 1$ **to** n **do**
    if *redundant(F, $S \cup \{\ell_i\}$)* **then**
      $S := S \cup \{\ell_i\}$;
  return S;
end

redundant$(F, S \cup \{l_i\})$ implements the check from Thm. 1 by using an SMT solver. It is important to note that the n SMT problems to be solved in the above loop share almost all of their clauses. For that reason we make use of an *incremental* SMT solver to solve this series of problems. An incremental SMT solver is able to profit from the similarity of the problems by transferring learned knowledge from one SMT solver call to the next (by means of learned conflict clauses). Experimental results in Sect. 4 indeed show the advantage of using an incremental SMT solver.

3.3   Removal of Redundant Linear Constraints

Suppose that formula (1) of Thm. 1 is unsatisfiable. Now we are looking for an efficient procedure to compute a boolean function G such that $G(\ell_{r+1}, \ldots, \ell_n)$

and $F(\ell_1, \ldots, \ell_n)$ represent the same predicates. Obviously, the boolean functions F and G do not need to be identical in order to achieve this objective; they are allowed to differ for 'inconsistent' arguments which can not be produced by evaluating the linear constraints with real values. The set of these arguments is described by the following set DC:

Definition 2. *The don't care set DC induced by linear constraints $\ell_1, \ldots, \ell_n$ is defined as $DC := \{(v_{\ell_1}, \ldots, v_{\ell_n}) \mid (v_{\ell_1}, \ldots, v_{\ell_n}) \in \{0,1\}^n$ and $\forall (v_{c_1}, \ldots, v_{c_f}) \in \mathbb{R}^f\ \exists 1 \leq i \leq n$ with $\ell_i(v_{c_1}, \ldots, v_{c_f}) \neq v_{\ell_i}\}$.*

As we will see in the following, it is possible to compute a function G as needed by making use of the don't care set DC. However, an efficient realization would certainly need a compact representation of the don't care set DC. Fortunately, a closer look at the problem reveals the following two interesting observations which turn our basic idea into a feasible approach:

1. In general, we do not need the complete set DC for the computation of the boolean function G.
2. A representation of a sufficient subset DC' of DC which is needed for removing the redundant constraints $\ell_1, \ldots, \ell_r$ is already computed by an SMT solver when checking the satisfiability of formula (1), if one assumes that the SMT solver uses the option of minimizing conflict clauses.

In order to explain how an appropriate subset DC' of DC is computed by the SMT solver (when checking the satisfiability of formula (1)) we start with a brief review of the functionality of an SMT solver:[3]

An SMT solver introduces constraint variables q_{ℓ_i} for linear constraints ℓ_i (just as in LinAIGs as shown in Fig. 1). First, the SMT solver looks for satisfying assignments to the boolean variables (including the constraint variables). Whenever the SMT solver detects a satisfying assignment to the boolean variables, it checks whether the assignment to the constraint variables is consistent, i.e., whether it can be produced by replacing real-valued variables by reals in the linear constraints. This task is performed by a linear program solver. If the assignment is consistent, then the SMT solver has found a satisfying assignment, otherwise it continues searching for satisfying assignments to the boolean variables. If some assignment $\epsilon_1, \ldots, \epsilon_m$ to constraint variables $q_{\ell_{i_1}}, \ldots, q_{\ell_{i_m}}$ was found to be inconsistent, then the boolean 'conflict clause' $(q_{\ell_{i_1}}^{\overline{\epsilon_1}} + \ldots + q_{\ell_{i_m}}^{\overline{\epsilon_m}})$ is added to the set of clauses in the SMT solver to avoid running into the same conflict again. The negation of this conflict clause describes a set of don't cares due to an inconsistency of linear constraints.

Now consider formula (1) which has to be solved by an SMT solver and suppose that the solver introduces boolean constraint variables q_{ℓ_i} for linear constraints ℓ_i and $q_{\ell_i'}$ for ℓ_i' $(1 \leq i \leq n)$. Whenever there is some satisfying assignment to boolean variables (including constraint variables) in the SMT solver, it will be necessarily shown to be inconsistent, since formula (1) is unsatisfiable.

---

[3] Here we refer to the *lazy* approach to SMT solving, see [18], e.g., for an overview.

In order to define an appropriate function G we introduce the concept of so-called orbits: For an arbitrary value $(v_{\ell_{r+1}}, \ldots, v_{\ell_n}) \in \{0,1\}^{n-r}$ the corresponding orbit is defined by

$$orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}) := \{(v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}) \mid (v_{\ell_1}, \ldots, v_{\ell_r}) \in \{0,1\}^r\}.$$

Now the following essential observations result from the unsatisfiability of formula (1): If some orbit $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ contains two different elements $v^{(1)} := (v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ and $v^{(2)} := (v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ with $F(v^{(1)}) \neq F(v^{(2)})$, then

(a) $v^{(1)} \in DC$ or $v^{(2)} \in DC$ and
(b) the SMT solver detects and records this don't care when solving formula (1).

In order to show fact (a), we consider the following assignment to the boolean abstraction variables in formula (1): Let $q_{\ell_1} := v_{\ell_1}, \ldots, q_{\ell_r} := v_{\ell_r}, q_{\ell'_1} := v'_{\ell_1}, \ldots,$ $q_{\ell'_r} := v'_{\ell_r}, q_{\ell_{r+1}} := q_{\ell'_{r+1}} := v_{\ell_{r+1}}, \ldots, q_{\ell_n} := q_{\ell'_n} := v_{\ell_n}$. (Thus $v^{(1)}$ is assigned to the abstraction variables for $\ell_1, \ldots, \ell_n$ and $v^{(2)}$ to the abstraction variables for $\ell'_1, \ldots, \ell'_n$.) It is easy to see that this assignment satisfies the boolean abstraction of formula (1). Since formula (1) is unsatisfiable, the assignment has to be inconsistent wrt. the interpretation of constraint variables by linear constraints. So there must be an inconsistency in the truth assignment to some linear constraints $\ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_n$. Since the linear constraints ℓ_i and ℓ'_i are based on disjoint sets of real variables $C = \{c_1, \ldots, c_f\}$ and $C' = \{c'_1, \ldots, c'_f\}$, already the partial assignment to $\ell_1, \ldots, \ell_n$ or the partial assignment to $\ell'_1, \ldots, \ell'_n$ has to be inconsistent, i.e., $v^{(1)} \in DC$ or $v^{(2)} \in DC$.

Fact (b) follows from the simple observation that the SMT solver has to detect and record the inconsistency of the assignment mentioned above in order to prove unsatisfiability of formula (1) and with minimization of conflict clauses it detects only conflicts which are confined either to $\ell_1, \ldots, \ell_n$ or to $\ell'_1, \ldots, \ell'_n$.[4]

Altogether this means that the elements of some $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ which are not in the subset DC' of DC computed by the SMT solver are either all mapped by F to 0 or are all mapped by F to 1. Thus, we can define an appropriate function G by don't care assignment as follows:

1. If $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}) \subseteq DC'$, then $G(v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ is chosen arbitrarily.
2. Otherwise $G(v_{\ell_{r+1}}, \ldots, v_{\ell_n}) = \delta$ with $F(orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}) \setminus DC') = \{\delta\}$, $\delta \in \{0,1\}$.

It is easy to see that G does not depend on variables $q_{\ell_1}, \ldots, q_{\ell_r}$ and that G is well-defined (this follows from $|F(orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}) \setminus DC')| = 1$), i.e., G is a possible solution according to Def. 1. This consideration also provides a proof for the 'if-part' of Thm. 1.

---

[4] For our purposes, it does not matter whether the inconsistency is given in terms of linear constraints $\ell_1, \ldots, \ell_n$ or $\ell'_1, \ldots, \ell'_n$. We are only interested in assignments of boolean values to linear constraints leading to inconsistencies; of course, the same inconsistencies will hold both for $\ell_1, \ldots, \ell_n$ and their copies $\ell'_1, \ldots, \ell'_n$.

A predicate dc which describes the don't cares in DC' may be extracted from the SMT solver as a disjunction of negated conflict clauses which record inconsistencies between linear constraints.

Note that according to case 1. of the definition above there may be several possible choices fulfilling the definition of G.

Redundancy Removal by Existential Quantification. A straightforward way of computing an appropriate function G relies on existential quantification:

- At first by $G' = F \wedge \overline{dc}$ all don't cares represented by dc are mapped to the function value 0.
- Secondly, we perform an existential quantification of the variables $q_{\ell_1}, \ldots, q_{\ell_r}$ in G': $G = \exists q_{\ell_1}, \ldots, q_{\ell_r} G'$. This existential quantification maps all elements of an orbit $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ to 1, whenever the orbit contains an element ϵ with $dc(\epsilon) = 0$ and $F(\epsilon) = 1$. Since due to the argumentation above there is no other element δ in such an orbit with $dc(\delta) = 0$ and $F(\delta) = 0$, G eventually differs from F only for don't cares defined by dc and it certainly does not depend on variables $q_{\ell_1}, \ldots, q_{\ell_r}$, i.e., existential quantification computes one possible solution for G according to Def. 1 (more precisely it computes exactly the solution for G which maps a minimum number of elements of $\{0,1\}^{n-r}$ to 1).

Redundancy Removal with Craig Interpolants. Although our implementation of LinAIGs supports quantification of boolean variables by a series of methods like avoiding the insertion of equivalent nodes, quantifier scheduling, BDD sweeping and node selection heuristics (see [17]), there remains the risk of doubling the representation size by quantifying a single boolean variable.[5] Therefore the computation of G by $G = \exists q_{\ell_1}, \ldots, q_{\ell_r} G'$ as shown above may potentially lead to large LinAIG representations (although it reduces the number of linear constraints).

On the other hand, this choice for G is only one of many other possible choices. Motivated by these facts we looked for an alternative solution. Here we present a solution which needs only one application of Craig interpolation [1,8,9,19] instead of a series of existential quantifications of boolean variables. Note that in this context Craig interpolation leads to an exact result (as one of several possible choices) and not to an approximation as in [9].

Don't cares can be assigned arbitrarily in order to make G independent from $q_{\ell_1}, \ldots, q_{\ell_r}$, thus our task is to find a boolean function $G(q_{\ell_{r+1}}, \ldots, q_{\ell_n})$ with

$$(F \wedge \overline{dc})(q_{\ell_1}, \ldots, q_{\ell_n}) \Longrightarrow G(q_{\ell_{r+1}}, \ldots, q_{\ell_n}), \tag{2}$$

$$G(q_{\ell_{r+1}}, \ldots, q_{\ell_n}) \Longrightarrow (F \vee dc)(q_{\ell_1}, \ldots, q_{\ell_n}). \tag{3}$$

Now let $A(q_{\ell_1}, \ldots, q_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n}, h_1, \ldots, h_l)$ represent the CNF for a Tseitin transformation [20] of $(F \wedge \overline{dc})(q_{\ell_1}, \ldots, q_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n})$ (with new auxiliary

---

[5] Basically, existential quantification of a boolean variable is reduced to a disjunction of both cofactors wrt. 0 and wrt. 1.

variables $h_1, \ldots, h_l$). Likewise, let $B(q'_{\ell_1}, \ldots, q'_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n}, h'_1, \ldots, h'_{l'})$ be the CNF for a Tseitin transformation of $(\overline{F} \wedge \overline{dc})(q'_{\ell_1}, \ldots, q'_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n})$ (with new auxiliary variables $h'_1, \ldots, h'_{l'}$ and new copies $q'_{\ell_1}, \ldots, q'_{\ell_r}$ of the variables $q_{\ell_1}, \ldots, q_{\ell_r}$).

Then A and B fulfill the precondition '$A \wedge B = 0$' for Craig interpolation [1,8]:

Suppose that there is a satisfying assignment to $A \wedge B$ given by $q_{\ell_1} := v_{\ell_1}, \ldots, q_{\ell_r} := v_{\ell_r}, q'_{\ell_1} := v'_{\ell_1}, \ldots, q'_{\ell_r} := v'_{\ell_r}, q_{\ell_{r+1}} := v_{\ell_{r+1}}, \ldots, q_{\ell_n} := v_{\ell_n}$, and the corresponding assignments to auxiliary variables $h_1, \ldots, h_l$ and $h'_1, \ldots, h'_{l'}$ which are implied by these assignments. According to the definition of A and B this would mean that the set $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ would contain two elements $(v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ and $(v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ which do not belong to the don't care set DC' and which fulfill $F(v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}) = 1$ and $F(v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}) = 0$. This is a contradiction to the property shown above that the elements of $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n})$ which are not in DC' are either all mapped by F to 0 or are all mapped by F to 1.

A Craig interpolant G computed for A and B has the following properties [1,8]:

- It depends only on common variables $q_{\ell_{r+1}}, \ldots, q_{\ell_n}$ of A and B,
- $A \Longrightarrow G$, i.e., G fulfills equation (2), and
- $G \wedge B$ is unsatisfiable, or equivalently, $G \Longrightarrow \overline{B}$, i.e., G fulfills equation (3).

This shows that a Craig interpolant for (A, B) is exactly one of the possible solutions for G which we were looking for. According to [8,9] a Craig interpolant can be computed in linear time based on a proof by resolution that a formula in CNF (in our case $A \wedge B$ as defined above) is unsatisfiable. Such proofs can be computed by any modern SAT solver with proof logging turned on.

4   Experimental Results

We implemented redundancy detection by incremental SMT solving and redundancy removal by Craig interpolation in the framework of LinAIGs. The implementation uses two SMT solvers via API calls. Yices [13] is used for all SMT solver calls except the generation of the don't care set. This means that Yices performs all equivalence checks needed for LinAIG compaction ([4,5], Sect. 2.1) and moreover, it is also used for the redundancy detection algorithm described in Sect. 3 in an incremental way. For the computation of the don't care set required for redundancy removal we use MathSAT [15], since it provides a method for extracting conflict clauses due to inconsistent assignments to linear constraints. The computation of the Craig interpolants is done with MiniSAT [21], where we made an extension to the proof logging version. All experiments were performed on an AMD Opteron with 2.6 GHz and 16 GB RAM under Linux.

Comparison of the LinAIG evolution with and without redundancy removal. In Fig. 4 we present a comparison of two typical runs of the model checker from [5]. The left diagram shows the evolution of the linear constraints over time and the right diagram shows the evolution of node counts. When we do not use
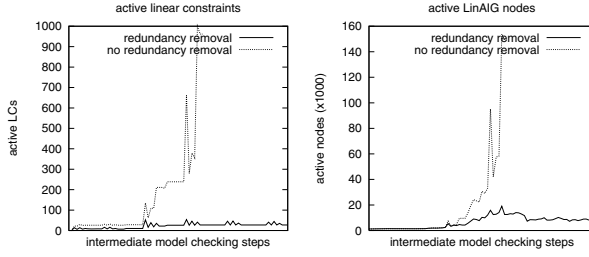
Fig. 4. Comparison of the LinAIG evolution with and without redundancy removal

redundancy removal, the number of linear constraint is quickly increasing up to 1000 and more, and the number of LinAIG nodes is exploding up to a value of 150,000. On the other hand, when using redundancy removal the number of linear constraints and the number of AIG nodes show only a moderate growth rate. This behavior has an immediate effect on the run times of the model checker: Whereas model checking finished within 15 minutes with redundancy removal, the version without redundancy removal did not finish within a timeout of 24 hours. This gives a strong evidence that redundancy removal is absolutely necessary when using quantifier elimination to keep the data structure compact in our model checking environment.

Existential quantification vs. Craig interpolation. Here we evaluate the two different approaches to removal of redundant constraints as presented in Sect. 3.3. The first approach uses existential quantification to eliminate redundant constraints, the second one uses our approach based on Craig interpolation. The benchmarks represent state sets extracted from the model checker from [5] during three runs with different model checking problems (the first group of runs was taken from an industrial case study considering a flap controller of an aircraft [5], the following two groups from the verification of collision avoidance protocols for train applications [22]). All these problems also contain boolean variables.

The results are given in Table 1. The numbers of AIG nodes and linear constraints before redundancy removal are shown in columns 2 and 3. In column 4 the detected number of redundant linear constraints is given. The times for the detection of redundancy and the don't care set generation are given in columns 5 and 6. Note that these values are the same for both approaches, because the difference lies only in the way linear constraints are actually removed. In the last four columns the results of the two algorithms are shown, where 'Δ nodes' denotes the difference between the number of AIG nodes before and after the removal step and 'time' is the CPU time needed for this step. We used a timeout of 7200 seconds and a memory limit of 4 GB.

The results clearly show that wrt. runtime the redundancy removal based on Craig interpolation outperforms the approach with existential quantification by far. Especially when the benchmarks are more complex and show a large number of redundant linear constraints, the difference between the two methods is substantial. Moreover, also the resulting AIG is often much smaller. It is interesting

Table 1. Comparison of redundancy removal: existential quantification vs. Craig interpolants

Benchmark	# AIG nodes	# linear constr.	# redundant lin. constr.	redundancy detection (s)	dc set creation (s)	RR exist. quant. Δ nodes	time (s)	RR Craig interp. Δ nodes	time (s)
stateset_1-1	1459	41	22	0.22	0.35	-541	7.19	-814	0.74
stateset_1-2	1716	74	27	0.51	0.71	-313	13.14	-1047	0.68
stateset_1-3	2340	105	22	1.89	2.41	459	25.35	-515	2.32
stateset_1-4	3500	142	28	8.02	5.53	1642	75.49	1062	10.08
stateset_1-5	2837	123	13	4.61	4.54	-230	12.34	1595	23.10
stateset_2-1	824	29	8	0.12	0.19	747	2.55	-142	0.43
stateset_2-2	1424	37	10	0.36	0.53	1104	3.45	233	1.19
stateset_2-3	3048	52	11	2.45	2.50	1996	10.13	171	4.22
stateset_2-4	1848	37	14	0.57	0.90	852	4.03	-149	1.34
stateset_3-1	495	74	44	0.10	0.14	656	4.55	-365	0.13
stateset_3-2	1775	297	228	1.86	1.74		>7200	-1453	1.60
stateset_3-3	6703	1281	1143	105.69	23.70		>7200	-5805	14.12
stateset_3-4	32021	5943	5706	2774.10	1012.78		>7200	-24633	126.19

to see that using incremental SMT solving techniques it was in many cases possible to detect large sets of redundant linear constraints in very short times. As shown in the previous experiment this pays off also in subsequent steps of model checking when quantifier elimination works on a representation with a smaller number of linear constraints. Considering column 6 we observe that run times for the generation of don't care sets by MathSAT are comparable to the run times of redundancy *detection*.[6]

Comparison of the LinAIG based quantifier elimination with other solvers. In a last experiment we compared our approach to quantifier elimination with several existing tools: REDLOG [12] is an extension to the computer algebra system REDUCE [11] and uses the quantifier elimination algorithm of Loos and Weispfenning [7], too, LIRA 1.1.2 [10] is an automata-based tool capable of representing sets of states over real, integer, and boolean variables, and CVC3 1.2.1 [14] as well as Yices 1.0.11 [13] are state-of-the-art SMT solvers.

The benchmarks formulas used in this experiment contain linear constraints and boolean variables, together with AND operators, negations, and quantifiers over real-valued variables. The formulas were extracted from the model checker [5] and represent continuous pre-image computations for state sets. All formulas contain two quantified variables, one is existentially quantified and the other one is universally quantified. All formulas are given in the SMT-LIB format [23] and are publicly available.[7] Since the SMT-LIB format only supports flat formulas (instead of shared graph structures), we had to confine ourselves to state set representations with moderate sizes.

For the SMT solvers we interpret free variables as implicitly existentially quantified and decide satisfiability, since they do not compute predicates representing all

---

[6] As already mentioned above, for technical reasons in our implementation we have to repeat the last step of redundancy detection (which actually was already performed by Yices) using MathSAT in order to be able to extract conflict clauses.

[7] http://abs.informatik.uni-freiburg.de/tacas09bench/

Table 2. Comparison of Solvers

Benchmark					LinAIG					REDUCE/REDLOG				LIRA		Yices		CVC3	
Name	AND	LC	B	R	N	AND	LC	B	Time	AND	LC	B	Time	Res.	Time	Res.	Time	Res.	Time
pre1	1K	22	5	4	27	30	12	5	1.48	830	26	5	0.14	SAT	27.71	?	0.03	?	0.22
pre2	2K	20	5	4	15	15	2	4	1.23	1133	35	5	0.21	SAT	67.98	?	0.05	?	0.25
pre3	5K	26	5	4	55	71	16	5	2.03	1918	39	5	0.43	SAT	443.66	?	0.15	?	0.40
pre4	160K	52	4	4	33	35	12	3	4.48	196483	224	4	38.06	timeout		?	3.94	?	4.65
pre5	188K	27	20	5	31	59	13	4	4.52	35356	42	16	14.67	memout		?	4.41	?	4.84
pre6	1396K	31	20	5	21	29	9	3	15.15	396887	68	20	98.89	timeout		?	32.88	?	22.81
pre7	3894K	30	20	5	32	111	8	4	46.58	memout				memout		?	148.47	?	90.97
pre8	6730K	44	20	5	186	545	14	12	68.95	memout				memout		?	239.20	?	111.56
pre9	9931K	52	8	4	555	8034	20	8	96.19	memout				memout		?	191.67	?	132.06

satisfying assignments. Our LinAIG based tool, REDLOG/REDUCE and LIRA additionally compute representations for predicates representing all satisfying assignments. Again, we used a time limit of 7200 CPU seconds and a memory limit of 4 GB for our experiments.

Table 2 shows the results. The first section 'Benchmark' shows details on the input formulas. The column 'AND' lists the number of AND operators in the formula, 'LC' lists the number of linear constraints, 'B' the number of boolean variables, and 'R' the number of real variables. The LinAIG section shows for the resulting predicates the numbers of AIG nodes ('N'), the numbers of AND operators in the corresponding flat formula ('AND'), the numbers of linear constraints ('LC'), boolean variables ('B'), and run times ('Time'). The run times include all CPU times necessary for reading the formulas, constructing LinAIGs, eliminating quantifiers, and removing redundant linear constraints. For the REDLOG tool we report the number of AND operators, linear constraints, and boolean variables in the resulting formula, as well as the run times needed for the computation. For LIRA, Yices and CVC3 run times are given together with the result whether the formula is proven to be satisfiable ('SAT') or unsatisfiable ('UNSAT'). The two SMT solvers Yices and CVC3 are not using a complete method and therefore may also report 'unknown' which is marked by '?'.

Our LinAIG based approach is able to eliminate the quantifiers of all formulas within a short runtime and moreover, returns formulas which are much more compact than the formulas produced by REDLOG/REDUCE, both in terms of AND operators and in terms of linear constraints. For mid-size examples the run times of REDUCE/REDLOG are outperformed by our tool, whereas the larger examples could not be solved by REDUCE/REDLOG. LIRA was able to solve only 3 out of 9 instances within the time or memory limit and it needs much more run time. The SMT solvers Yices and CVC3 were not able to solve any of the examples. Note however that these solvers are not restricted to the subclass of formulas we consider in this paper. They are able to handle the more general AUFLIRA class of formulas [24] and for handling formulas with quantifiers they make use of heuristics based on E-matching [25] which are not tuned to problems that contain only linear arithmetic.

In summary, the experiments clearly demonstrate that for the subclass of formulas considered in this paper we were able to provide an efficient method both wrt. run times and wrt. the sizes of the resulting formulas.

5 Conclusions and Future Work

We presented an approach for optimizing non-convex polyhedra based on the removal of redundant constraints. Our experimental results show that our approach can be successfully applied to solving quantified formulas including linear real arithmetic and boolean formulas. The method is based on an elaborate scheme for keeping graph-based representations of intermediate results as compact as possible, with redundancy removal as an essential component. Since our method does not only solve satisfiability of formulas, but constructs predicates of all satisfying assignments to the free variables in the formula, our results may suggest to use the presented method in the future also as a fast preprocessor for more general formulas by simplifying subformulas from the subclass considered in this paper. Moreover, it will be interesting to apply the methods to underlying theories different from linear real arithmetic, too.

Acknowledgements

References

1. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Journal on Symbolic Logic 22(3), 269–285 (1957)
2. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer 1(1–2), 110–122 (1997)
3. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
4. Damm, W., Disch, S., Hungar, H., Pang, J., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Automatic verification of hybrid systems with large discrete state space. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 276–291. Springer, Heidelberg (2006)
5. Damm, W., Disch, S., Hungar, H., Jacobs, S., Pang, J., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact state set representations in the verification of linear hybrid systems with large discrete state space. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 425–440. Springer, Heidelberg (2007)

6. Wang, F.: Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. IEEE Trans. on Software Engineering 31(1), 38–52 (2005)
7. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
8. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal on Symbolic Logic 62(3), 981–998 (1997)
9. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
10. Eisinger, J., Klaedtke, F.: Don't care words with an application to the automata-based approach for real addition. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 67–80. Springer, Heidelberg (2006)
11. Griss, M.L.: The reduce system for computer algebra. In: ACM 1975: Proceedings of the 1975 annual conference, pp. 261–262. ACM Press, New York (1975)
12. Dolzmann, A., Sturm, T.: Redlog: computer algebra meets computer logic. SIGSAM Bull. 31(2), 2–9 (1997)
13. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
14. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A cooperating validity checker. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 500–504. Springer, Heidelberg (2002)
15. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATH-SAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
16. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.K.: FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley (2005)
17. Pigorsch, F., Scholl, C., Disch, S.: Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling. In: FMCAD, pp. 89–96 (2006)
18. Sebastiani, R.: Lazy satisability modulo theories. JSAT 3, 141–224 (2007)
19. Lee, C.C., Jiang, J.H.R., Huang, C.Y., Mishchenko, A.: Scalable exploration of functional dependency by interpolation and incremental SAT solving. In: ICCAD, pp. 227–233 (2007)
20. Tseitin, G.: On the complexity of derivations in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logics (1968)
21. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 541–638. Springer, Heidelberg (2004)
22. Damm, W., Mikschl, A., Oehlerking, J., Olderog, E.-R., Pang, J., Platzer, A., Segelken, M., Wirtz, B.: Automating verification of cooperation, control, and design in traffic applications. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 115–169. Springer, Heidelberg (2007)
23. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2 (2006), `http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`
24. Barrett, C., Deters, M., Oliveras, A., Stump, A.: Satisfiability Modulo Theories Competition (SMT-COMP) 2008: Rules and Precedures (2008), `http://smtcomp.org/rules08.pdf`
25. Detlefs, D., Nelson, G., Saxe, J.: Simplify: A theorem prover for program checking. J. ACM 52(3), 365–473 (2005)

All-Termination$(T)^\star$

Panagiotis Manolios and Aaron Turon

Northeastern University
{pete,turon}@ccs.neu.edu

Abstract. We introduce the ALL-TERMINATION(T) problem: given a termination solver T and a collection of functions F, find every subset of the formal parameters to F whose consideration is sufficient to show, using T, that F terminates. An important and motivating application is enhancing theorem proving systems by constructing the set of strongest induction schemes for F, modulo T. These schemes can be derived from the set of *termination cores*, the minimal sets returned by ALL-TERMINATION(T), without any reference to an explicit measure function. We study the ALL-TERMINATION(T) problem as applied to the size-change termination analysis (SCT), a PSPACE-complete problem that underlies many termination solvers. Surprisingly, we show that ALL-TERMINATION(SCT) is also PSPACE-complete, even though it substantially generalizes SCT. We develop a practical algorithm for ALL-TERMINATION(SCT), and show experimentally that on the ACL2 regression suite (whose size is over 100MB) our algorithm generates stronger induction schemes on 90% of multiargument functions.

1 Introduction

Reasoning about recursion requires induction. But there may be several induction schemes that apply to a given recursive function, and different theorems may require the use of different induction schemes. Finding induction schemes for a given function is particularly important for automated theorem provers that perform induction heuristically. In this context, Boyer and Moore explored the strong relationship between termination and both recursion and induction [1]. They showed that proving termination is the key to justifying function definitions and induction schemes, and developed methods for doing so mechanically. This was one of the major insights that led to the success of the Boyer-Moore family of theorem provers, which includes ACL2 [2].

In this paper, we introduce a generalization of the classic termination problem: ALL-TERMINATION. The motivating application for this problem is its use in mechanically deriving and justifying as many induction schemes for a function as possible, using methods like Boyer and Moore's. Each induction scheme is closely tied to the pattern of recursion in the function, so the schemes are likely to be useful in automated reasoning about the function.

We begin with a few examples, first using traditional induction schemes, and then describing the schemes we can derive through ALL-TERMINATION analysis. Consider the function even, which determines whether a natural number is even:

```
even n = if n = 0 then T else if n = 1 then F else even (n - 2)
```

To show the correctness of even, there are a few induction principles we might apply. An obvious first choice is standard induction over the naturals. However, this principle does not suffice for proving the theorem, because it is not possible to prove that $\mathtt{even}(n)$ is correct assuming only that $\mathtt{even}(n-1)$ is correct; we need instead that $\mathtt{even}(n-2)$ is correct. On the other hand, we could employ strong induction on the naturals. We would then have to show

$$\langle \forall m < n :: \mathtt{even}(m) \text{ iff } \lfloor m/2 \rfloor = m/2 \rangle \implies (\mathtt{even}(n) \text{ iff } \lfloor n/2 \rfloor = n/2)$$

where n is implicitly universally quantified, a shorthand we will use throughout this section. While this is a reasonable choice of induction scheme, it is a bit *ad hoc*. In particular, it is hard to see how to derive such an induction scheme from the definition of even in an automated way.

What Boyer and Moore propose instead is to derive an induction scheme from the pattern of *recursion* in the function body. For the even function, we can derive following induction scheme for proving $\langle \forall n :: \varphi(n) \rangle$:

$$\varphi(0), \qquad \varphi(1), \qquad n \neq 0, n \neq 1, \varphi(n-2) \implies \varphi(n)$$

How do we know that the induction scheme is sound? By proving that even *terminates* on all inputs. Since the induction scheme corresponds directly to the recursion of even, knowing that the recursion terminates allows us to apply well-founded induction and soundly derive the induction scheme above. Notice that the induction scheme can be applied to *any* φ, not just φ involving even. But the derivation of the scheme was mechanically guided by the definition of even.

Now we turn to a more interesting example: the function zip, which takes a pair of lists and produces a list of pairs:

```
zip xs ys = if nil?(xs) or nil?(ys) then nil
            else cons (head x, head y) (zip (tail xs) (tail ys))
```

Recall that a *measure* μ on a function f is another function, on the same domain, that maps into a well-ordered structure[1] such that whenever $f(a)$ calls $f(b)$, we have $\mu(a) > \mu(b)$. Because the successive values of μ cannot decrease infinitely, f cannot recur infinitely. We say a set P of formal parameter names for a function is *measurable* if there exists a measure on that function that uses only those arguments. Suppose length measures the length of a list. For zip, the sets $\{\mathtt{xs}, \mathtt{ys}\}$, $\{\mathtt{xs}\}$, and $\{\mathtt{ys}\}$ are measurable, because $\mathtt{length(xs)} + \mathtt{length(ys)}$, $\mathtt{length(xs)}$, and $\mathtt{length(ys)}$, respectively, are measures. Ignoring the measures themselves, we can use measurable sets of a function to derive induction schemes. For instance, here are two induction schemes for zip:

---

[1] A set with a total order that has no infinite descending chains $x_1 > x_2 > \cdots$.

Measurable set: $P = \{\text{xs}, \text{ys}\}$ Measurable set: $P = \{\text{xs}\}$

1. $\varphi(\texttt{[]}, \texttt{ys})$ 1. $\varphi(\texttt{[]}, \texttt{ys})$
2. $\varphi(\texttt{xs}, \texttt{[]})$ 2. $\varphi(\texttt{xs}, \texttt{[]})$
3. $\varphi(\texttt{xs}, \texttt{ys}) \implies \varphi(\texttt{x:xs}, \texttt{y:ys})$ 3′. $\langle \forall \texttt{zs} :: \varphi(\texttt{xs}, \texttt{zs}) \rangle \implies \varphi(\texttt{x:xs}, \texttt{y:ys})$

The intuition is that, if a parameter like `ys` does not appear in a measurable set, then that parameter can vary freely without affecting termination; hence, the parameter can be *instantiated* freely during induction without invalidating the induction scheme. The induction scheme based on $\{\text{xs}\}$ is *stronger* than the one for $\{\text{xs}, \text{ys}\}$: any theorem proved using the latter can be proved using the former, but not *vice versa*.

In practice, Boyer-Moore theorem provers use complex heuristics to propose an induction scheme that is likely needed to prove a given theorem. The proposed scheme must then be *justified*. The key point is that, just as with the simple examples above, the justification is based on measurable sets. Users can introduce new measurable sets, but only through a manual process involving a termination proof. Our goal is to automate the process of justifying induction schemes by computing *all* the measurable sets for a given function.

We thus define ALL-TERMINATION as follows: given a recursive function f, find its measurable sets. The ALL-TERMINATION problem is a generalization of the classic termination decision problem: a program is terminating iff it has at least one measurable set. Therefore, ALL-TERMINATION is undecidable. However, decades of work on termination have yielded powerful, but decidable, termination analyses. For any such termination analysis, T, we can pose the ALL-TERMINATION(T) problem: given a function f and a termination solver, T, find as many measurable sets as possible using T.

In this paper, we focus on the size-change termination analysis (SCT [3]) because several powerful termination analyses depend on it (see Section 5 for examples). An introduction to the size-change framework is given in Section 3. Then, in Section 4, we study ALL-TERMINATION(SCT) in detail and show that its complexity is the *same* as the complexity of SCT: they are both PSPACE-complete problems. We also develop an algorithm, using *dual-horn minimization*. We have implemented this algorithm on a prototype basis, and executed it on the ACL2 regression suite, consisting of over 11,000 functions. We found that over 90% of multiargument functions have at least one measurable set that was smaller than the full set of arguments to the function, and 7% of the multi-argument functions had multiple, incomparable measurable sets. These results suggest that ALL-TERMINATION can increase automation in theorem provers.

An important practical consideration is the tension between termination analysis and ALL-TERMINATION analysis. Since termination analysis tends to be expensive, and theorem provers often require functions to be shown terminating before they can be admitted, the goal is to decide termination as quickly as possible, using the simplest analysis [4]. On the other hand, we can get better ALL-TERMINATION results by employing heavyweight methods, even when simpler methods suffice to show termination—but this involves more work. The algorithm we develop takes this tension into account and is *responsive*: it

answers the basic termination question first, without incurring any additional overhead. Only after termination is settled does it proceed with the full ALL-TERMINATION(T) analysis. This approach allows a theorem prover to use spare CPU cycles or cores to detect new induction schemes in the background, after the function is determined to terminate. It also allows the theorem prover to use ALL-TERMINATION(T) analysis in a demand-driven way, asking for induction schemes when the need arises.

A version of this paper with more detail and full proofs is available online [5].

2 All-Termination(T)

We postulate a universe of programs PROG, but do not specify a particular syntax or semantics. Intuitively, a program $F \in$ PROG is a mutually-recursive nest of functions; F terminates iff each function in F terminates on every input. Formally, we require that for every program $F \in$ PROG, there is a corresponding transition system $\mathcal{C}_F$, called the *semantic call graph* of F, which terminates iff F does and whose states are function names with actual arguments. Given universes of function names $\mathcal{F}$, parameter names $\mathcal{P}$, and values $\mathcal{V}$, we say:

Definition 1. *A **semantic call graph** $\mathcal{C}$ is a pair $(S, \rightarrow)$ with $S \subseteq \mathcal{F} \times (\mathcal{P} \rightharpoonup \mathcal{V})$ the set of **states** and $\rightarrow \subseteq S \times S$ the **transition relation**. The elements of $\mathcal{P} \rightharpoonup \mathcal{V}$ are the partial functions from $\mathcal{P}$ to $\mathcal{V}$.*

Definition 2. *A semantic call graph $\mathcal{C}$ is **terminating** if it contains no infinite sequence of transitions $s_1 \rightarrow s_2 \rightarrow \cdots$.*

The Fibonacci function and its semantic call graph are:

```
fib 0 = 1
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

fib(0) fib(1) fib(2) fib(3) $\cdots$
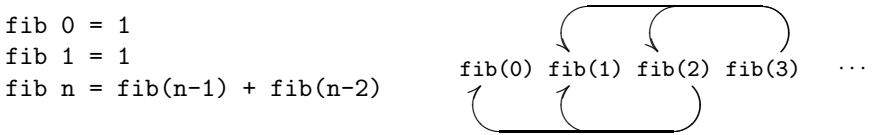
A semantic call graph records the actual function calls made in order to compute a given function application. In general, $\mathcal{C}_F$ is an infinite, undecidable structure: even determining whether there is a transition between two states is undecidable. We can express termination of semantic call graphs in terms of measure functions, the standard tool for proving termination, as follows.

Proposition 1. *$(S, \rightarrow)$ is terminating iff there exists a well-ordered set $(W, >)$ and a **measure** μ, i.e., a map $\mu : S \rightarrow W$ such that if $s \rightarrow t$ then $\mu(s) > \mu(t)$.*

This proposition follows from basic results in set theory, showing that every terminating relation can be extended to a well-order and that every well-order is order-isomorphic to a unique ordinal number.

If (f, V) is a state in a semantic call graph $\mathcal{C}$, the values of the formal arguments in $\mathrm{dom}(V)$ are the observations available to a measure on $\mathcal{C}$. Thus, to restrict the arguments a measure can observe, and thereby force it to ignore certain arguments, we restrict the domain of V:

Definition 3. *Given $V : \mathcal{P} \rightharpoonup \mathcal{V}$, $f \in \mathcal{F}$ and $P \subseteq \mathcal{P}$, we define the* **restrictions**

$$(V \restriction P)(x) = \begin{cases} V(x) & x \in dom(V) \cap P, \\ \text{undefined} & \textit{otherwise} \end{cases} \qquad (f, V) \restriction P = (f, V \restriction P)$$

Informally, a set of formal parameter names P is measurable if there is a measure that "uses" only those arguments. We can formalize this idea using restriction.

Definition 4. *P is a* **measurable set** *for $\mathcal{C} = (S, \rightarrow)$ if there exists a measure $\mu : S \rightarrow W$ such that, if $s, t \in S$ and $s \restriction P = t \restriction P$, then $\mu(s) = \mu(t)$.*

Note that if $\mathcal{C}$ has any measurable set, then in particular $\mathcal{C}$ is terminating. Termination analyses are usually formulated so that they imply the termination of a program, but not the existence of any particular measurable set. To define ALL-TERMINATION(T), we need to limit the analysis T to "use" only a certain set of formal parameters, just as for measures.

Definition 5. *A* **termination analysis** *T is a predicate such that, if $T(F, P)$, then P is a measurable set for $\mathcal{C}_F$.*

Finally, given a termination analysis T and a program F, the termination cores of F modulo T are the minimal P such that $T(F, P)$. That is,

$$\text{ALL-TERMINATION}(T)(F) = \min\{P \subseteq \mathcal{P} \ : \ T(F, P)\}$$

3   The Size-Change Framework

The semantic call graph $\mathcal{C}_F$ precisely captures the recursive behavior of F, at the cost of undecidability. A fruitful approach to termination analysis is to consider safe approximations of $\mathcal{C}_F$. This section describes the *size-change framework* of Lee, Jones, and Ben-Amram [3]. The main result, Theorem 1, is not new. However, our presentation of the framework includes some innovations that are needed for studying ALL-TERMINATION(SCT): we give a fuller account of the connection between SCT and semantic call graphs (including a notion of simulation), and we make explicit the notion of evaluation.

Example. Consider the well-known total function `ack`:

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Traditionally, to prove that `ack` terminates, a measure μ is introduced corresponding to a lexicographic order on the arguments. The size-change framework takes an alternative perspective, focusing on the change in size of each argument independently, without having to concoct a single measure on the tuple of arguments. We first observe that in every recursive call to `ack`, either the first

argument decreases, or the first argument does not increase while the second decreases. We display this size-change data as follows:

$$G_1: \quad \boxed{\begin{array}{c} m \xrightarrow{\ >\ } m \\[2mm] n \qquad n \end{array}} \qquad\qquad G_2: \quad \boxed{\begin{array}{c} m \xrightarrow{\ \geq\ } m \\[2mm] n \xrightarrow{\ >\ } n \end{array}}$$

It follows that any putative infinite recursion would involve an infinite sequence of argument size changes of the form above—but we can show that this is not possible. If size change G_1 occurs infinitely often, then m decreases infinitely, which is impossible under a well-order. Otherwise, since we are considering an *infinite* sequence of size changes, it must be that size change G_2 occurs uninterrupted as an infinite suffix of the sequence. But then n decreases infinitely, which is again impossible. Hence, `ack` terminates. The size-change framework reformulates such reasoning into a decidable analysis.

For simplicity, we postulate a single well-ordering $>$ on all values in $\mathcal{V}$.[2] The notion of size-change "data" above is formalized into a structure called a *size-change graph*. An *annotated call graph* (ACG) is a directed graph with function names as nodes, and an edge from f to g for each call to g that occurs in the body of f. The edges of an ACG are labeled by size-change graphs, which record the size relationship between the arguments of f and g. More formally, we write

$$\begin{array}{lll} p, q, r \in \text{LAB} & = \{>, \geq\} & \text{size-change label} \\ G, H \in \text{SCG} & = 2^{\mathcal{P} \times \text{LAB} \times \mathcal{P}} & \text{size-change graph} \\ \mathcal{G}, \mathcal{H} \in \text{ACG} & = 2^{\mathcal{F} \times \text{SCG} \times \mathcal{F}} & \text{annotated call graph} \end{array}$$

We write $x \xrightarrow{r} y$ for $(x, r, y) \in G$ and $f \xrightarrow{G} g$ for $(f, G, g) \in \mathcal{G}$. We also sometimes write $G \in \mathcal{G}$ for $f \xrightarrow{G} g$ if the function names f and g are unimportant.

The annotated call graph for `ack` is: $G_1 \left(\; \text{ack} \;\right) G_2$. The intuitive demonstration that `ack` terminates was based on sequences of argument size changes during recursive function calls. A potential sequence of function calls is just a path through an ACG.

Definition 6. *A **multipath** π through an ACG $\mathcal{G}$ is a (potentially infinite) sequence of edges from $\mathcal{G}$, connected at nodes: $\pi = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \xrightarrow{G_3} \cdots$.*

We write $\mathcal{G}^\omega$ for the set of nonempty multipaths over $\mathcal{G}$ and $\mathcal{G}^+$ for the set of finite, nonempty ones. We sometimes write $G_1, G_2, \ldots$ or $\langle G_i \rangle$ to describe a multipath when the function names are irrelevant.

The reason $\pi = \langle G_i \rangle$ is a *multipath* and not just a path is that the elements G_i of the sequence are themselves graph structures. In particular, a multipath may contain many *threads* through its size-change graphs.

Definition 7. *A **thread** in a multipath $\pi = \langle G_i \rangle$ is a sequence of size-change edges $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ such that $x_{i-1} \xrightarrow{r_i} x_i \in G_i$ for all $i > 0$.*

---

[2] Multiple orders can also be handled [4].

For example, consider the multipath $\texttt{ack} \xrightarrow{G_1} \texttt{ack} \xrightarrow{G_2} \texttt{ack} \xrightarrow{G_1} \texttt{ack}$ in $\mathcal{G}_{\texttt{ack}}$. Its only thread is $m \xrightarrow{>} m \xrightarrow{\geq} m \xrightarrow{>} m$. On the other hand, the multipath $\texttt{ack} \xrightarrow{G_2} \texttt{ack} \xrightarrow{G_2} \texttt{ack}$ has two threads: $m \xrightarrow{\geq} m \xrightarrow{\geq} m$ and $n \xrightarrow{>} n \xrightarrow{>} n$.

Threads track a given value as it flows through the arguments of successive function calls. A value being tracked by a thread can never increase, but it must decrease any time it passes through a >-labeled size-change edge. Size-change termination analysis works by considering all potential infinite multipaths through an ACG, and demonstrating that each of them contains an infinite thread that forces its value to decrease infinitely often. By well-foundedness, such infinite decreases cannot occur, and so all infinite multipaths are ruled out.

Definition 8

(1) An infinite thread $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ has **infinite descent** if $r_i \; = \; >$ for infinitely-many i.

(2) A multipath π has **infinite descent** if it has a thread with infinite descent.

(3) $\mathcal{G}$ is **size-change terminating** if every infinite multipath $\pi \in \mathcal{G}^\omega$ has a suffix with infinite descent.

We have motivated ACGs in terms of function calls, but it remains to formally connect ACGs to semantic call graphs. An ACG $\mathcal{G}$ can be seen as a finite description of a semantic call graph $\mathcal{C}_\mathcal{G}$ that relates states according to the possible size changes given in $\mathcal{G}$:

Definition 9. The **semantic call graph determined by** $\mathcal{G}$ is $\mathcal{C}_\mathcal{G} = (S, \rightarrow)$, where $S = \mathcal{F} \times (\mathcal{P} \rightharpoonup \mathcal{V})$ and

$$(f, V) \rightarrow (g, U) \quad \textit{iff} \quad \langle \exists f \xrightarrow{G} g \in \mathcal{G} :: \langle \forall x \xrightarrow{r} y \in G :: V(x) \; r \; U(y) \rangle \rangle$$

In order to use the size-change termination of $\mathcal{G}$ to show the termination of F, we must relate $\mathcal{C}_\mathcal{G}$ and $\mathcal{C}_F$. The relation we use is a form of *simulation*.

Definition 10. Given two semantic call graphs $\mathcal{C}_1 = (S_1, \rightarrow_1)$ and $\mathcal{C}_2 = (S_2, \rightarrow_2)$, a **simulation** between $\mathcal{C}_1$ and $\mathcal{C}_2$ is a relation $R \subseteq S_1 \times S_2$ such that

- for each s_1 there is some s_2 with $s_1 \; R \; s_2$
- if $s_1 \; R \; s_2$ then $s_1 = (f, V)$ and $s_2 = (f, U)$ with $f \in \mathcal{F}$ and $U = V \restriction dom(U)$
- if $s_1 \; R \; s_2$ and $s_1 \rightarrow_1 s_1'$ then there exists an s_2' such that $s_2 \rightarrow_2 s_2'$ and $s_1' \; R \; s_2'$

We say $\mathcal{C}'$ **simulates** $\mathcal{C}$, written $\mathcal{C} \sqsubseteq \mathcal{C}'$, if there exists a simulation R between $\mathcal{C}$ and $\mathcal{C}'$. Intuitively, if $\mathcal{C}'$ simulates $\mathcal{C}$, then $\mathcal{C}'$ admits at least as many behaviors as $\mathcal{C}$. We say $\mathcal{G}$ is **safe for** F if $\mathcal{C}_F \sqsubseteq \mathcal{C}_\mathcal{G}$. In general, finding a safe ACG $\mathcal{G}$ for a program F is difficult, and is a problem that the size-change framework does not address (but see [4]). For our purposes, it is sufficient to postulate a function $analyze : \text{PROG} \rightarrow \text{ACG}$ such that $analyze(F)$ is safe for F.

The next two propositions allow us to conclude that if $\mathcal{G}$ is safe for F and $\mathcal{G}$ is size-change terminating then F terminates.

Proposition 2. $\mathcal{G}$ *is size-change terminating iff* $\mathcal{C}_\mathcal{G}$ *is terminating.*

Proposition 3. *If* $\mathcal{C} \sqsubseteq \mathcal{C}'$ *and* $\mathcal{C}'$ *is terminating then* $\mathcal{C}$ *is terminating.*

Deciding size-change termination for an ACG $\mathcal{G}$ is a PSPACE-complete problem, but the standard algorithm used in practice needs exponential space in the worst case [3]; we present this algorithm next.

Suppose $\mathcal{G}$ is an ACG. If $f_0 \xrightarrow{G_1} \cdots \xrightarrow{G_n} f_n$ is a multipath in $\mathcal{G}^+$, we know that according to $\mathcal{G}$, a call to f_0 could result in a call to f_n. But what can we say about the size of the arguments to f_n in terms of the arguments to f_0? What we want is a way to compose size-change graphs along a multipath.

Definition 11. *We define composition of size-change labels and graphs by*

$$p \cdot q = \begin{cases} \geq & p = \geq \text{ and } q = \geq \\ > & \text{otherwise.} \end{cases} \qquad G \cdot H = \{x \xrightarrow{p \cdot q} z \; : \; x \xrightarrow{p} y \in G, y \xrightarrow{q} z \in H\}$$

Definition 12. *The* **evaluation** *of* $\pi = \langle G_1, \ldots, G_n \rangle \in \mathcal{G}^+$ *is* $[\![\pi]\!] = G_1 \cdot \cdots \cdot G_n$.

Note that composition is associative, so evaluation is well-defined. The evaluation of a multipath π is useful because it compactly summarizes the threads in π:

Proposition 4. $x \xrightarrow{r} y \in [\![\pi]\!]$ *iff there exists a thread* $\langle x \xrightarrow{r_1} z_1 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} z_{n-1} \xrightarrow{r_n} y \rangle$ *in* π, *with* $r = r_1 \cdot \cdots \cdot r_n$.

The key step for the size-change termination algorithm is to compute the *closure* of an annotated call graph $\mathcal{G}$ under composition: the set $\{[\![\pi]\!] \mid \pi \in \mathcal{G}^+\}$. The closure is formally defined as a least fixpoint. The algorithm looks for certain "maximal" size-change graphs in the closure, called *idempotents*.

Definition 13.

(1) The **closure** *of* $\mathcal{G}$ *under* $\cdot$ *is the smallest set satisfying*

$$cl(\mathcal{G}) = \mathcal{G} \cup \{f \xrightarrow{G \cdot H} h \; : \; f \xrightarrow{G} g, g \xrightarrow{H} h \in cl(\mathcal{G})\}$$
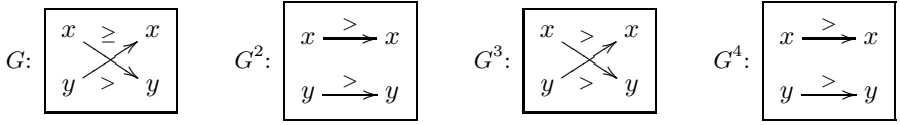
(2) A size-change graph G *is* **idempotent** *if* $G \cdot G = G$.

Theorem 1 (Lee *et al.* [3]). $\mathcal{G}$ *is size-change terminating iff for every* $f \xrightarrow{G} g \in cl(\mathcal{G})$ *such that* G *is idempotent, there is an edge* $x \xrightarrow{>} x \in G$.

Example. Consider the following function `perm`, which permutes its two arguments, decreasing one of them, until one of them is zero.

```
perm 0 y = y
perm x 0 = x
perm x y = perm (y - 1) x
```

How can we use the theorem above to show that `perm` terminates? First, we need to construct $\mathcal{G}_{\texttt{perm}}$. Since there is only one recursive call in `perm`, $\mathcal{G}_{\texttt{perm}}$ has only one node and one edge. The size-change graph G for `perm` and its powers are:

$$G: \quad \begin{array}{c} x \;\geq\; x \\ \times \\ y \;\xrightarrow{\;>\;}\; y \end{array} \qquad G^2: \quad \begin{array}{c} x \xrightarrow{\;>\;} x \\ y \xrightarrow{\;>\;} y \end{array} \qquad G^3: \quad \begin{array}{c} x \;\xrightarrow{\;>\;}\; x \\ \times \\ y \;\xrightarrow{\;>\;}\; y \end{array} \qquad G^4: \quad \begin{array}{c} x \xrightarrow{\;>\;} x \\ y \xrightarrow{\;>\;} y \end{array}$$

Note that G^2 is idempotent, so $G^4 = G^2$. Consequently, the only distinct size-change graphs in $cl(\mathcal{G}_{\texttt{perm}})$ are G, G^2, and G^3. Since G^2 has an edge $x \xrightarrow{\;>\;} x$, and G^2 is the only idempotent graph in $cl(\mathcal{G}_{\texttt{perm}})$, $\mathcal{G}_{\texttt{perm}}$ is size-change terminating.

The standard algorithm for deciding size-change termination is based on Theorem 1: compute $cl(analyze(F))$ as a least fixpoint, and check the strict self-edge condition on the idempotent elements. To adapt this algorithm for all-termination, we will record some additional information as size-change graphs are composed, and build a constraint system from this information after the algorithm finishes. The minimal solutions to these constraints will be exactly the termination cores of F.

4 All-Termination(SCT)

Recall that a termination analysis is a predicate $T(F, P)$ that holds only if P is a measurable set for F. Thus, the first step in studying ALL-TERMINATION(SCT) is to formulate such a predicate $SCT(F, P)$, with the property that $\langle \exists P :: SCT(F, P) \rangle$ holds exactly when F is size-change terminating. Size-change analysis, like many termination analyses, does not explicitly construct a measure witnessing termination; it only implies the existence of one.[3] We need a way to restrict size-change analysis to a set of parameters P, such that this implied measure only uses parameters from P. To do this, we derive an ACG $analyze(F) \upharpoonright P$ whose size-change termination implies that P is a measurable set of F. This restricted ACG simply drops any size-change information not related to parameters in P.

Definition 14. *Given G, $\mathcal{G}$, and P, we define the **restrictions***

$$G \upharpoonright P = \{x \xrightarrow{r} y \in G \;:\; x, y \in P\} \qquad \mathcal{G} \upharpoonright P = \{f \xrightarrow{G \upharpoonright P} g \;:\; f \xrightarrow{G} g \in \mathcal{G}\}$$

Similarly, we introduce a notion of restriction on semantic calls graphs, which will allow us to derive a useful new characterization of measurable sets.

Definition 15. *Given $\mathcal{C} = (S, \rightarrow)$ and P, we define the **restriction***

$$\mathcal{C} \upharpoonright P = \big(\{(f, V \upharpoonright P) \;:\; (f, V) \in S\}, \rightsquigarrow\big)$$

where $s \rightsquigarrow t$ iff there exist $s', t' \in \mathcal{C}$ such that $s = s' \upharpoonright P$, $t = t' \upharpoonright P$, and $s' \rightarrow t'$.

---

[3] It is possible to effectively construct a measure from size-change analysis, and thereby extract a *single* measurable set, but the size of the measure is exponential [6].

Proposition 5. *For all $\mathcal{C}$, $\mathcal{G}$ and $P \subseteq \mathcal{P}$ we have*

(1) P is a measurable set for $\mathcal{C}$ iff $\mathcal{C} \restriction P$ is terminating,
(2) $\mathcal{C} \sqsubseteq \mathcal{C} \restriction P$,
(3) if $\mathcal{C} \sqsubseteq \mathcal{C}'$ then $\mathcal{C} \restriction P \sqsubseteq \mathcal{C}' \restriction P$, and
(4) $\mathcal{C}_{\mathcal{G}} \restriction P \approx \mathcal{C}_{\mathcal{G} \restriction P}$, where $(\approx) = (\sqsubseteq) \cap (\sqsupseteq)$.

We now have all the pieces needed to define the *SCT* predicate:

$$SCT(F, P) \iff analyze(F) \restriction P \text{ is size-change terminating}$$

Theorem 2. *SCT is a termination analysis.*

Proof. Using Proposition 5, $\mathcal{C}_F \restriction P \sqsubseteq \mathcal{C}_{analyze(F)} \restriction P \approx \mathcal{C}_{analyze(F) \restriction P}$. If $SCT(F, P)$ holds then $analyze(F) \restriction P$ is size-change terminating, so $\mathcal{C}_{analyze(F) \restriction P}$ is terminating, and hence so is $\mathcal{C}_F \restriction P$. By Proposition 5, P is a measurable set of F.

Deciding $SCT(F, P)$ is PSPACE-complete, since the restriction of an ACG to P can be computed in polynomial time, after which the problem reduces to size-change termination. Somewhat surprisingly, ALL-TERMINATION(SCT) has the *same* complexity.

Theorem 3. ALL-TERMINATION(SCT) *is* PSPACE-*complete.*

Proof. ALL-TERMINATION(SCT) is PSPACE-hard because $\langle \exists P :: SCT(F, P) \rangle$ can be reduced to ALL-TERMINATION(SCT)(F) in constant space by executing ALL-TERMINATION(SCT)(F) until it either halts with no output or produces its first output, and $\langle \exists P :: SCT(F, P) \rangle$ is PSPACE-hard. On the other hand, the following algorithm solves ALL-TERMINATION(SCT) in polynomial space:

ALL-TERMINATION(SCT)(F)
    for $P \subseteq \mathcal{P}$ do
        if $SCT(F, P)$ and $\langle \forall Q \subset P :: SCT(F, Q) = \mathsf{False} \rangle$ then output P

The algorithm uses polynomial space because $SCT(F, P)$ is in PSPACE, and all of the loops can be implemented using counters whose size is logarithmic in the size of $2^{\mathcal{P}}$, hence linear in the size of $\mathcal{P}$, where $\mathcal{P}$ is the set of parameters to functions in F.

This theorem generalizes to any PSPACE-complete termination analysis. Having settled the basic complexity question, we now turn to practical considerations. The algorithm above executes SCT at least $2^{|\mathcal{P}|}$ times. The algorithm we introduce below runs SCT once, gathering information from which it extracts the termination cores. The key to the algorithm is understanding the threads and multipaths in $(\mathcal{G} \restriction P)^+$ in terms of those in $\mathcal{G}^+$. We first observe that each multipath in $(\mathcal{G} \restriction P)^+$ is the *restriction* of a multipath in $\mathcal{G}^+$, as follows.

Definition 16. *Given a multipath $\pi = f_0 \xrightarrow{G_1} \cdots \xrightarrow{G_n} f_n$ in $\mathcal{G}^+$, the **restriction** of π to $P \subseteq \mathcal{P}$ is $\pi \restriction P = f_0 \xrightarrow{G_1 \restriction P} \cdots \xrightarrow{G_n \restriction P} f_n$, which is a multipath in $(\mathcal{G} \restriction P)^+$.*

Proposition 6. *Let $P \subseteq \mathcal{P}$.*

(1) If $\pi \in (\mathcal{G} \restriction P)^+$ then there exists a $\pi' \in \mathcal{G}^+$ such that $\pi = \pi' \restriction P$.
(2) If $\pi \in \mathcal{G}^+$ then the threads of $\pi \restriction P$ are exactly the threads $\langle x_0 \xrightarrow{r_1} \cdots \xrightarrow{r_n} x_n \rangle$ of π such that each x_i is in P.

Notice that as size-change graphs are composed, some information about the possible threads within them is lost. For example, if $x \xrightarrow{\geq} z \in G \cdot H$, we know that there is *some* y for which $x \xrightarrow{p} y \in G$ and $y \xrightarrow{q} z \in H$ with $pq = \geq$, but given only the composed graph $G \cdot H$ it is not possible to determine which choices of y would suffice. More generally, given a multipath $\pi \in \mathcal{G}^+$, we can determine all of its threads; but, given only $x \xrightarrow{r} y \in [\![\pi]\!]$, the most we can say is that there is *some* thread in π from x to y (by Proposition 6). Thus, if we want to reason about the threads of $\pi \restriction P$ (and hence the edges in $[\![\pi \restriction P]\!]$) in terms of $[\![\pi]\!]$, we need to keep track of which variables contribute to each edge $x \xrightarrow{r} y \in [\![\pi]\!]$. We do this using *annotated size-change graphs*:

$$\mathbb{G}, \mathbb{H} \in \text{ASCG} = 2^{\mathcal{P} \times (\text{LAB} \times 2^{\mathcal{P}}) \times \mathcal{P}} \quad \text{annotated size-change graphs}$$

Intuitively, if an edge $x \xrightarrow[Q]{r} y$ is in an ASCG $\mathbb{G}$, then there is some thread relating x to y with size-change r, involving at most the parameters in Q.

If G is a size-change graph in $\mathcal{G}$, and $x \xrightarrow{r} y \in G$, the only parameters needed to show that there is a thread from x to y are x and y themselves. Thus we have a simple way of producing initial ASCGs from SCGs:

Definition 17. *The ASCG for G is $\lfloor G \rfloor = \{x \xrightarrow[\{x,y\}]{r} y \; : \; x \xrightarrow{r} y \in G\}$.*

Just as with SCGs, we have composition, evaluation, and closure for ASCGs.

Definition 18

(1) Annotated composition and evaluation are defined as follows:

$$\mathbb{G} \odot \mathbb{H} = \{x \xrightarrow[P \cup Q]{pq} z \; : \; x \xrightarrow[P]{p} y \in \mathbb{G}, \; y \xrightarrow[Q]{q} z \in \mathbb{H}\}$$

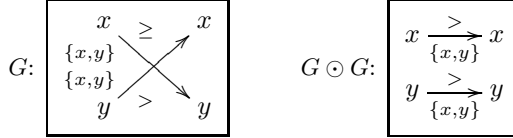$$\lfloor G_1, \ldots, G_n \rfloor = \lfloor G_1 \rfloor \odot \cdots \odot \lfloor G_n \rfloor$$

*(2) The **annotated closure** of $\mathcal{G}$ under $\odot$ is the least set satisfying*

$$acl(\mathcal{G}) = \{f \xrightarrow{\lfloor G \rfloor} g \; : \; f \xrightarrow{G} g \in \mathcal{G}\} \cup \{f \xrightarrow{\mathbb{G} \odot \mathbb{H}} h \; : \; f \xrightarrow{\mathbb{G}} g, g \xrightarrow{\mathbb{H}} h \in acl(\mathcal{G})\}$$

We can now reason about the multipaths of $(\mathcal{G} \restriction P)^+$ in terms of $\mathcal{G}^+$:

Proposition 7. *Let $\pi \in \mathcal{G}^+$. Then $x \xrightarrow{r} y \in [\![\pi \restriction P]\!]$ iff there exists a $Q \subseteq P$ such that $x \xrightarrow[Q]{r} y \in \lfloor \pi \rfloor$.*

Example. Returning to the `perm` example, we ask: is {x} a measurable set for `perm`? No: a function taking only the x parameter for `perm` cannot possibly be a measure. To see why, consider that `perm 1 2` calls `perm 1 1`. Thus, a measure μ for `perm` using only x would have to have the property that $\mu(1) > \mu(1)$ which is clearly impossible. A similar argument shows that {y} is not a measurable set. We can now reanalyze the `perm` function in using annotated size-change graphs, to see how they are used to discover that {x} and {y} are not measurable sets. We begin with the same graph G we had before, but with annotated edges.

$$G: \quad \begin{array}{c} x \xrightarrow{\ \geq\ } x \\[-2pt] {\scriptstyle \{x,y\}} \\[-2pt] {\scriptstyle \{x,y\}} \\[-2pt] y \xrightarrow{\ >\ } y \end{array} \qquad\qquad G \odot G: \quad \begin{array}{c} x \xrightarrow[\{x,y\}]{>} x \\[4pt] y \xrightarrow[\{x,y\}]{>} y \end{array}$$

As before, $G \odot G$ is idempotent. The annotations on the edges of $G \odot G$, however, tell us that to justify a decrease from, *e.g.*, x to x in $G \odot G$, we *must* consider the formal argument y as well.

The next result shows that we have completely characterized size-change termination for any $\mathcal{G} \restriction P$ in terms of the annotated closure $acl(\mathcal{G})$.

Theorem 4. $\mathcal{G} \restriction P$ *is size-change terminating iff for every* $f \xrightarrow{\mathbb{G}} g \in acl(\mathcal{G})$ *such that* $\mathbb{G}$ *is idempotent, there is an edge* $x \xrightarrow{>}_{Q} x \in \mathbb{G}$ *with* $Q \subseteq P$.

Corollary 1. *Let* $\mathcal{I} = \{\mathbb{G} \in acl(analyze(F)) : \mathbb{G} \ idempotent\}$. *We have* ALL-TERMINATION$(SCT)(F) = \{P \subseteq \mathcal{P} : \langle \forall \mathbb{G} \in \mathcal{I} :: \langle \exists x \xrightarrow{>}_{Q} x \in \mathbb{G} :: Q \subseteq P \rangle \rangle\}$.

We can use Corollary 1 as the basis for an algorithm as follows. First, compute $acl(analyze(F))$ as a least fixpoint, and extract the set of idempotent ASCGs as $\mathcal{I}$. Then, for each $\mathbb{G} \in \mathcal{I}$, construct the constraint $\bigvee_{x \xrightarrow{>}_{Q} x \in \mathbb{G}} \bigwedge_{y \in Q} y$. The

collection (conjunction) of these constraints is a constraint system Φ_F whose solutions are the elements of ALL-TERMINATION(SCT). It turns out that by introducing variables, this constraint system can be expressed in *dual-horn* form.

This is a useful observation because there is an *output-sensitive* algorithm for enumerating the minimal solutions to dual-horn formulas [7]. Output sensitivity means that the running time of the algorithm is bounded by the number of *outputs* it produces, and more-over provides "pay-as-you-go" enumeration. For dual-horn minimization, the time is *exponential* in the number of outputs; moreover, the constraint system Φ_F may have an exponential number of minimal solutions. In practice, however, functions have very few termination cores (no more than 3 in our experiment), whereas they often have many arguments (sometimes more than 20 in our experiment). Hence we much prefer the annotation-based algorithm (exponential in the former) than the naive algorithm (exponential in the latter). Finally, we have developed a version of dual-horn minimization based on incremental SAT-solving, which we hope will perform well when faced with a larger number of cores.

The algorithm described above has another appealing property: it can be made *responsive*, by which we mean it can answer the basic termination problem as quickly as the standard size-change algorithm. If the program cannot be shown to terminate, there is no need to continue. If termination is established, then responsiveness can be exploited by the theorem prover in various ways, including the following two. First, the theorem prover can run the ALL-TERMINATION(T) algorithm to completion. For interactive theorem proving applications, this can be done using spare CPU cycles (*e.g.*, by using an underutilized CPU core), because the user is free to continue as soon as termination has been established, and any new induction schemes found can be quietly recorded by the theorem prover. Second, the theorem prover can suspend the ALL-TERMINATION(T) algorithm, coming back to it only when it needs new induction schemes, thereby using the analysis in a demand-driven way. Responsiveness is obtained by controlling the least fixpoint computation of $acl(analyze(F))$ so that the size change graphs needed to compute $cl(analyze(F))$ are generated first. This process differs from the basic size-change algorithm only in that we record the size-change graph annotations required for the annotated closure. Once termination is established, the fixpoint computation for the annotated closure proceeds.

Experimental Results

We have implemented our ALL-TERMINATION(SCT) algorithm in ACL2, an industrial-strength theorem proving system. Our implementation served as a new back-end for the *calling context graph* (CCG) termination analysis, which is implemented in the ACL2 Sedan [4,8]. Normally, CCG analysis uses SCT as a back-end; by using ALL-TERMINATION(SCT) instead, we are able to determine the measurable sets for a function. ACL2 has a large regression suite, with over 11,000 function definitions, each of which must be proved terminating in order to be admitted into the logic. The regression suite is particularly appealing because it arises from the work of researchers around the world, with examples ranging from bit-vector libraries used by AMD, to set theory libraries, graph algorithms and model checkers. In short, the code in the regression suite provides a large, realistic sample of ACL2 programs.

We executed our analysis on the entire regression suite The time running ALL-TERMINATION(SCT) was negligible compared to the time spent within CCG's static analysis, which involves theorem proving. We collected data on the 1,728 recursive, multiargument functions in the suite. More than 90% of the functions had at least one termination core that did not include all the arguments to the function, and about 7% of the functions had more than one termination core. These findings attest to the utility of ALL-TERMINATION: the 90% of functions with nontrivial termination cores can be given a stronger induction scheme, using ALL-TERMINATION, than would otherwise be possible. Thus, by generating stronger induction schemes, our analysis has the potential to extend the automation provided by theorem provers [1,2].

5   Related Work

The termination problem dates back to Turing, who called it the "Printing Problem" [9], and there has been steady interest in termination ever since. Here we can only briefly touch upon the work most directly related to ours.

Boyer and Moore's work [1], developing the strong relationship between termination and both recursion and induction the context of automated theorem proving, provided the impetus for the work we have presented. The idea of ALL-TERMINATION, too, can be traced back to Boyer and Moore [1]. However, the approach they used to find measurable subsets just iterates over their termination analysis in the naive way: it has exponential complexity and little in common with the work presented here, beyond the initial motivation. We know of no other work studying ALL-TERMINATION.

Termination analysis is currently an active area of research. There is much interest in termination in the context of term-rewrite systems and logic programs [10,11,12,13]. There is also interest in proving termination of programs written in imperative languages, such as C. This work tends to focus on semi-algebraic functions, whose termination behavior is governed by integer arithmetic. Most of it has been even more narrowly defined, dealing only with systems whose behavior is linear [14], though there are extensions to programs with polynomial behavior [15]. Also, abstraction-refinement has been applied to termination analysis, and subsequently used to find bugs in device drivers [16].

This paper has focused on size-change termination analysis [3], which was introduced in the setting of an applicative language and has since served as a framework for several other analyses. This includes work on termination in term-rewrite systems that combines size-change analysis with the dependency pair method and recursive path orderings [12]. Tools based on these ideas include APrOVE [11]. Another example is work on calling context graphs and measures, which is used to prove termination of functional programs [4], and has been implemented in ACL2s [8] and Isabelle [17].

Recently, the problem of conditional termination has been studied [18]. While we are interested in how we can add behaviors to programs while maintaining termination, conditional termination asks how to remove behaviors to ensure termination. This leads to the obvious question: what about ALL-CONDITIONAL-TERMINATION(T)? Similarly, the non-termination problem [19] gives rise to the ALL-NON-TERMINATION(T) problem.

6   Conclusions and Future Work

We introduced the ALL-TERMINATION(T) problem and analyzed the complexity when T is size-change (SCT) analysis. We showed that ALL-TERMINATION(SCT) is a PSPACE-complete problem, and introduced an algorithm for solving it. The algorithm imposes no overhead on solving the basic termination problem, and it can be used to generate a subset of all possible termination cores (up to a user provided bound). We implemented our algorithm in ACL2 and ran it on the ACL2 regression suite, consisting of over 100MB of code developed over several decades by a

worldwide user base. Our experiments showed that on over 90% of multiargument functions in the regression suite, we were able to provide stronger induction schemes than those obtained with size change analysis. Our primary focus for future work is analyzing the ALL-TERMINATION(T) problem for other termination analyses.

Acknowledgements. Alec Heller gave helpful feedback on several drafts.

References

1. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, London (1979)
2. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Dordrecht (2000)
3. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92. ACM Press, New York (2001)
4. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 401–414. Springer, Heidelberg (2006)
5. Manolios, P., Turon, A.: All-Termination(T). Technical Report NU-CCIS-09-01, Northeastern University (2009)
6. Ben-amram, A.M., Lee, C.S.: Ranking functions for size-change termination II. In: RDP-WST (2007)
7. Ben-Eliyahu, R., Dechter, R.: On computing minimal models. Annals of Mathematics and Artificial Intelligence 18, 3–27 (1996)
8. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: The ACL2 Sedan. ENTCS 174(2), 3–18 (2007)
9. Turing, A.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society 42(2), 230–265 (1936)
10. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 210–220. Springer, Heidelberg (2004)
12. Thiemann, R., Giesl, J.: Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen (January 2003)
13. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. The Journal of Logic Programming 41(1), 103–123 (1999)
14. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004)
15. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)
16. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426. ACM Press, New York (2006)
17. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
18. 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)
19. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL, pp. 147–158. ACM, New York (2008)

Ground Interpolation for the Theory of Equality

Alexander Fuchs[1], Amit Goel[2], Jim Grundy[2], Sava Krstić[2], and Cesare Tinelli[1]

[1] Department of Computer Science, The University of Iowa
[2] Strategic CAD Labs, Intel Corporation

Abstract. Given a theory $\mathcal{T}$ and two formulas A and B jointly unsatisfiable in $\mathcal{T}$, a *theory interpolant* of A and B is a formula I such that (i) its non-theory symbols are shared by A and B, (ii) it is entailed by A in $\mathcal{T}$, and (iii) it is unsatisfiable with B in $\mathcal{T}$. Theory interpolants are used in model checking to accelerate the computation of reachability relations. We present a novel method for computing ground interpolants for ground formulas in the theory of equality. Our algorithm computes interpolants from colored congruence graphs representing derivations in the theory of equality. These graphs can be produced by conventional congruence closure algorithms in a straightforward manner. By working with graphs, rather than at the level of individual proof steps, we are able to derive interpolants that are pleasingly simple (conjunctions of Horn clauses) and smaller than those generated by other tools.

1 Introduction

The *Craig Interpolation Theorem* [4] asserts—for every inconsistent pair of first-order formulas A, B—the existence of a formula I that is implied by A, inconsistent with B, and written using only logical symbols and symbols that occur in both A and B.

Analogues of this result hold for a variety of logics and logic fragments. Recently, they have found practical use in symbolic model checking. Applications, starting with the work by McMillan [7], involve computation of interpolants in propositional logic or in quantifier-free logics with (combinations of) theories such as the theory of equality, linear real arithmetic, arrays, and finite sets [8,14,6,3]. There are now techniques that use interpolants to obtain property-driven approximate reachability sets or transition relations, and also to compute refinements for predicate abstraction. Experimental results show that interpolation-based techniques are often superior to previous ones.

An important functionality in much of this work is the computation of ground interpolants in the *theory of equality*, also known as the theory of *uninterpreted functions* (*EUF*). The ground interpolation algorithm for this theory used in existing interpolation-based model checkers was developed by McMillan [8]. It derives interpolants from proofs in a formal system that contains rules for the basic properties of equality.

In this paper, we present a novel method for ground *EUF* interpolation. We compute interpolants from *colored congruence graphs* that compactly represent

EUF derivations from two sets of equalities, and can be produced in a straight-forward manner by conventional congruence closure algorithms. Working with graphs makes it possible to exploit the global structure of proofs in order to streamline the interpolant generation. Our interpolants are conjunctions of Horn clauses, the simplest conceivable form for this theory. In most cases, they are smaller and logically simpler than those produced by McMillan's method.

Our interpolation algorithm is described and proved correct in §5. In §4, we give a series of examples to highlight important aspects of the algorithm. Its inherent game-like structure is explained at a high level in §3. A detailed comparison with McMillan's method is given in §6, together with experimental data on a set of benchmarks derived from the SMT-LIB suite.

2   Ground Theory Interpolation

Interpolation is a property of fragments of logical theories. To state it for a fragment $\mathcal{F}$, we need know only the following:

- a partition of symbols used to build formulas in $\mathcal{F}$ into *logical* and *non-logical*
- the *entailment relation* $A \models B$ between formulas in $\mathcal{F}$

Let $\mathcal{F}(X)$ be the set of all formulas in $\mathcal{F}$ whose non-logical symbols belong to X. By definition, $\mathcal{F}$ has the INTERPOLATION PROPERTY if for every $A \in \mathcal{F}(X)$ and $B \in \mathcal{F}(Y)$ such that $A, B \models$ false, there exists $I \in \mathcal{F}(X \cap Y)$ such that $A \models I$ and $B, I \models$ false. The formula I will be called the INTERPOLANT for the pair A, B. Note the asymmetry: $\neg I$ is an interpolant for the pair B, A.

The classic Craig's theorem says that the set of first-order logic formulas has the interpolation property. (The non-logical symbols are the predicate and function symbols, and variables.) It also implies a "modulo theory" generalization, where, for a given first-order theory $\mathcal{T}$ over a signature Σ, $\mathcal{F}$ is the set of all Σ-formulas, $\models$ is the $\mathcal{T}$-entailment, and the symbols of Σ are treated as logical. The case where $\mathcal{T}$ and Σ are empty is Craig's original theorem.

Of particular interest is the interpolation property for quantifier-free fragments of theories. It may or may not hold, depending on the theory. Take, for example, the quantifier-free fragment of integer arithmetic, and let $A = \{x = 2u\}, B = \{x = 2v + 1\}$. A and B are inconsistent in this theory, and the formula $(\exists u)(x = 2u)$ is an interpolant. However, there is no quantifier-free interpolant.

By definition, a theory has the GROUND INTERPOLATION PROPERTY if its quantifier-free fragment has the interpolation property. Aside from *EUF*, several other theories of interest in model checking have this property [6].

If we want an efficient algorithm for ground $\mathcal{T}$-interpolation, it suffices to have one that works for inputs A and B that are *conjunctions of ground literals*. To see this, refer to [8,3] for a description of a general mechanism to combine such a restricted interpolation procedure with a method for computing interpolants in propositional logic [12,7].

Example 1. The sets of inequalities $A = \{3x - z - 2 \leq 0, -2x + z - 1 \leq 0\}$ and $B = \{3y - 4z + 12 \leq 0, -y + z - 1 \leq 0\}$ are mutually inconsistent, as

witnessed by the linear combination with *positive* coefficients $2 \cdot (3x - z - 2 \leq 0) + 3 \cdot (-2x + z - 1 \leq 0) + 1 \cdot (3y - 4z + 12 \leq 0) + 3 \cdot (-y + z - 1 \leq 0)$ that simplifies to $2 \leq 0$. The A-part of this linear combination $2 \cdot (3x - z - 2 \leq 0) + 3 \cdot (-2x + z - 1 \leq 0)$ gives us the interpolant $I = (z - 7 \leq 0)$ for A, B. Generalizing what goes on in this example, one can obtain a ground interpolation procedure for the linear arithmetic with real coefficients. See, e.g., [3].

3   Interpolation as a Cooperative Game

There is a way to compute interpolants in a theory $\mathcal{T}$ as a cooperative game between two deductive provers for $\mathcal{T}$—possibly two copies of the same prover. We give an informal description of the *ground interpolation game*. The same idea applies with minor modifications to other interpolation problems, but the success of the method is not guaranteed in general.

Let A and B be two sets of ground formulas such that $A \cup B$ is $\mathcal{T}$-unsatisfiable. We allow A and B to contain *free symbols*, i.e., predicate and function symbols not in the signature Σ of $\mathcal{T}$. Let Σ_I be the *shared signature*, the expansion of Σ with the free symbols occurring in both A and B.

The interpolation game. The participants are the A-*prover* and the B-*prover*. The game starts with $S_A = S_B = \varnothing$ and proceeds so that at each turn one of the following happens:

- the A-prover adds to S_A a new ground Σ_I-clause C such that $A, S_B \models_{\mathcal{T}} C$
- the B-prover adds to S_B a new ground Σ_I-clause C such that $B, S_A \models_{\mathcal{T}} C$

The game ends when the B-prover could add false to S_B. An interpolant for A and B can then be computed as follows.

Let C be a clause of S_A and let $C_1, \ldots, C_n$ be the clauses of S_B actually used by the A-prover to derive C. Let us call $C_1, \ldots, C_n$ the B-PREMISES of C and call the formula $C_1 \wedge \cdots \wedge C_n \Rightarrow C$ the A-JUSTIFICATION of C. Similarly, we can define the set A-PREMISES of each clause $C \in S_B$. For $C \in S_B$, define the CUMULATIVE SET OF PREMISES $\mathcal{P}(C)$ recursively by

$$\mathcal{P}(C) = \{C\} \cup \bigcup \{\mathcal{P}(D) \mid D \text{ is a } B\text{-premise of an } A\text{-premise of } C\}$$

It can be shown that the conjunction of A-justifications of A-premises of clauses in $\mathcal{P}(\mathsf{false})$ is an interpolant for A, B.

For some theories with the ground interpolation property, the game admits *complete strategies*, guaranteed to end the game when A and B are jointly unsatisfiable in the theory. Depending on the theory, these strategies can be considerably more restrictive in the choice of clauses to propagate from one prover to the other. For instance, by results of [13], when the theory is Σ-*convex*,[1] it is enough to propagate just positive unit clauses in all rounds of the game except

---

[1] A theory is CONVEX if $L \models_{\mathcal{T}} p_1 \vee \cdots \vee p_k$ implies $L \models_{\mathcal{T}} p_i$ for some i, where L is any set of ground literals and the p_i are any positive literals.

possibly the last. In that case, all interpolants computed will be conjunctions of Horn clauses.

The interpolation method we describe in §5.5 for the theory of equality can be understood as an implementation of this interpolation game, with clause propagation restricted to (positive) equalities. The method does not literally use two provers; it rather takes a proof of unsatisfiability of $A \cup B$, treats it as if cooperatively generated by an A-prover and a B-prover, and extracts an interpolant from it as sketched above.

Remark 1. Nelson and Oppen's method for combining decision procedures [9] is similar to the interpolation game. The main differences are that for the Nelson-Oppen procedure (i) the input sets of formulas A_1 and A_2 need not be jointly $\mathcal{T}$-unsatisfiable; (ii) the goal is not to produce interpolants for A_1 and A_2 but just to witness the $\mathcal{T}$-unsatisfiability of $A_1 \cup A_2$; (iii) $\mathcal{T}$ is the union of two signature-disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$; (iv) each formula A_i is built from the symbols of $\mathcal{T}_i$ and free constants; (v) each A_i-prover works just over $\mathcal{T}_i$ instead of the whole $\mathcal{T}$; (vi) additional restrictions on $\mathcal{T}_1$ and $\mathcal{T}_2$ guarantee termination when $A_1 \cup A_2$ is $\mathcal{T}$-satisfiable. A description of a Nelson-Oppen combination framework in terms similar to our interpolation game can be found in [5].

4   *EUF* Interpolation Examples

Let us fix the terminology first. A ground *EUF* term is either a free constant or (inductively) an application $f(t_1, \ldots, t_n)$ of an n-ary function symbol f to terms $t_1, \ldots, t_n$. An *EUF* literal is an equality $s = t$ between terms, or the negation $s \neq t$ of it (a disequality). We use $=$ also to write equalities at the meta-level, relying on the context to disambiguate. For convenience, we treat all equalities modulo symmetry: an equality $s = t$ will stand indifferently for $s = t$ or $t = s$.

Example 2. The picture in Figure 1(a) demonstrates the inconsistency of $A = \{z_1 = x_1,\ x_1 = x_2,\ z_2 = x_2,\ x_2 = f(z_3),\ f(z_3) = x_3,\ x_3 = z_4\}$ and $B = \{z_1 = y_1,\ y_1 = f(z_2),\ f(z_2) = y_2,\ y_2 = z_3,\ z_3 = y_3,\ y_2 \neq z_4\}$ that follows by transitivity of equality. The interpolant is the equality $z_1 = z_4$ that summarizes the A-chain. For the variation in Figure 1(b), the interpolant is the conjunction $z_1 = z_2 \land f(z_3) = z_4 \land f(z_2) = z_3$ of summaries of A-chains. For yet another variation, modify Figure 1(b) by moving the disequality sign to the edge $\langle x_3, z_4 \rangle$. The interpolant changes to $z_1 = z_2 \land f(z_3) \neq z_4 \land f(z_2) = z_3$. In light of the interpolation game (§3) where the A-prover and the B-prover alternate in deriving sets of equalities, we can see that in all these example the game can end after the second round: A generates some implied literals in the shared language, and B is inconsistent with their conjunction. (Here and elsewhere in this section, a *round* of the game is a sequence of steps done by the same prover.)

Example 3. When the inconsistency of $A \cup B$ requires congruence reasoning, an interpolant in the form of a conjunction of equalities need not exist. Let $A = \{u_1 = x \cdot u_0,\ v_1 = x \cdot v_0\}$ and $B = \{u_0 = v_0,\ u_1 \neq v_1\}$. (The dot is an infix
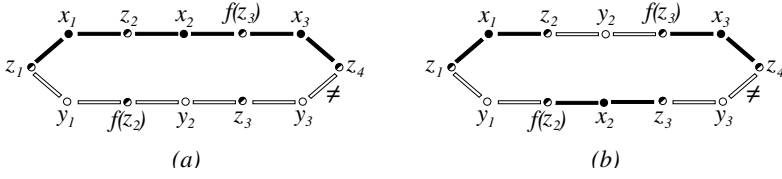
Fig. 1. Transitivity chains with dark (light) edges representing A-literals (B-literals). For the vertex coloring convention, see Example 9.

function symbol.) There are no equalities implied by A that do not contain x. The transitivity chain $u_1 = x \cdot u_0 = x \cdot v_0 = v_1$ contradicts $u_1 \neq v_1 \in B$, but its middle equality is not implied by A. However, A does imply it under the condition $u_0 = v_0$ that B provides. That gives us the interpolant $u_0 = v_0 \Rightarrow u_1 = v_1$. The game can take 3 rounds in which $u_0 = v_0$ is derived first (by B), then $u_1 = v_1$ (by A), then false (by B).

Example 4. Generalizing the previous example, consider this matrix-organized set of literals:

$$u_0 = v_0 \qquad \begin{matrix} x_1 \cdot u_0 = u_1 \\ x_1 \cdot v_0 = v_1 \end{matrix} \qquad \begin{matrix} x_2 \cdot u_1 = u_2 \\ x_2 \cdot v_1 = v_2 \end{matrix} \qquad \cdots \qquad \begin{matrix} x_n \cdot u_{n-1} = u_n \\ x_n \cdot v_{n-1} = v_n \end{matrix} \qquad u_n \neq v_n$$

Let A be the set of equalities occurring in the odd-numbered columns (count columns starting from *one*!) of this matrix, and B be the set of the remaining equalities; see Figure 2. The shared symbols are $u_0, v_0, \ldots, u_n, v_n$, the symbols local to A are $x_2, x_4, \ldots$, and the symbols local to B are $x_1, x_3, \ldots$.

The game takes $n + 2$ rounds. It begins with the A-prover adding $u_0 = v_0$ to S_A. Then, using the equalities from the second column, the B-prover can derive $u_1 = v_1$, and add it to S_B. Now, the A-prover can use this equality together with equalities from the third column to derive $u_2 = v_2$ and add it to S_A. Assuming n is even, the last equality $u_n = v_n$ will be derived by the A-prover, after which B derives false. Collecting justifications of all equalities derived by A, we obtain the interpolant

$$(u_0 = v_0) \wedge (u_1 = v_1 \Rightarrow u_2 = v_2) \wedge \cdots \wedge (u_{n-1} = v_{n-1} \Rightarrow u_n = v_n).$$

Example 5. With $A = \{x = z_1,\, x \cdot z_2 = z_3\}$ and $B = \{y = z_2,\, z_1 \cdot y \neq z_3\}$, pictured in Figure 3, we can derive inconsistency from the chain $z_3 = x \cdot z_2 = z_1 \cdot y \neq z_3$, where the congruence reasoning that produces the middle equality uses an equality from A and an equality from B ($x = z_1$ and $z_2 = y$), and cannot be derived by either A or B alone. A simple split of the problematic equality into two produces a chain in which every literal follows from either A or B: $z_3 = x \cdot z_2 = z_1 \cdot z_2 = z_1 \cdot y \neq z_3$. The summary $z_3 = z_1 \cdot z_2$ of the A-chain is our interpolant. The upshot is that creating an interpolant may require terms (in this case $z_1 \cdot z_2$) that do not occur in either A or B. See Lemma 1 below.
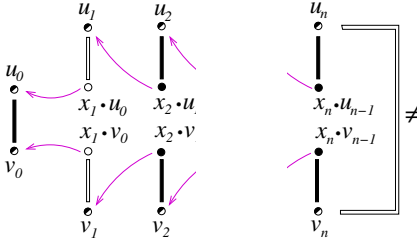
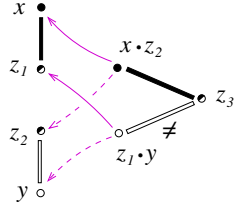**Fig. 2.** A long derivation. (Example 4)

Fig. 3. Who derives $x \cdot x_2 = z_1 \cdot y$? (Example 5)

5   Interpolants From Congruence Closure

5.1   Congruence Closure

Satisfiability of sets of *EUF* literals can be determined by the CONGRUENCE CLOSURE ALGORITHM. The algorithm takes as inputs a finite subterm-closed set T of ground terms and a finite set E of ground equalities. Its state is an *undirected* graph Γ, initialized so that its vertex set is T and its edge set is empty. We write $u \sim v$ to mean that u and v are connected by a path in Γ. The algorithm proceeds as follows.

(**cc1**) Choose $s, t \in T$ such that $s \not\sim t$ and either
    (**a**) $(s = t) \in E$; or
    (**b**) s is $f(s_1, \ldots, s_k)$, t is $f(t_1, \ldots, t_k)$, and $s_1 \sim t_1$, $\ldots$, $s_k \sim t_k$.
    Then add the edge $\langle s, t \rangle$ to Γ.
(**cc2**) Repeat (**cc1**) for as long as possible.

Theorem 1. [10,11] *Let $\sim$ be the equivalence relation obtained by running the congruence closure algorithm above. For every $s, t \in T$, one has $E \models s = t$ if and only if $s \sim t$. Moreover, the set $E \cup \{s \neq t \mid s \not\sim t\}$ is satisfiable.* $\square$

Let L be an arbitrary set of ground *EUF* literals. We have $L = L_= \cup L_{\neq}$, where the literals in $L_=$ and $L_{\neq}$ are equalities and disequalities respectively. To check whether L is satisfiable, it suffices to run the congruence closure algorithm with $E = L_=$ and T being the set of all terms occurring in L. By Theorem 1, L is satisfiable if and only if $s \not\sim t$ holds for every disequality $s \neq t$ in $L_{\neq}$. Note that L is unsatisfiable if and only if $L_= \cup \{\delta\}$ is unsatisfiable for some $\delta \in L_{\neq}$—the convexity property of *EUF*.

5.2   Congruence Graphs

Define a CONGRUENCE GRAPH over E to be any intermediate graph Γ obtainable by the congruence closure algorithm above. The assumption $s \not\sim t$ in (**cc1**) ensures that every congruence graph is acyclic. Thus, if $u \sim v$ in a congruence

graph Γ, then there is a unique PATH connecting them. This path is denoted $\overline{uv}$. EMPTY PATHS are those of the form $\overline{uu}$.

The edges of Γ introduced by step (**cc1-a**) will be called BASIC; those introduced in step (**cc1-b**) are DERIVED. A derived edge $\langle f(u_1, \ldots, u_k), f(v_1, \ldots, v_k) \rangle$ has k PARENT PATHS $\overline{u_1 v_1}, \ldots, \overline{u_k v_k}$, some (but not all) of which may be empty.

Example 6. Each of the graphs in Figures 1–3, *when we delete from it the edge marked with the $\neq$ symbol*, is a congruence graph over the appropriate set of equalities $(A \cup B)_=$. All edges in these graphs are basic; the arrows indicate where derived edged can be added.

Example 7. Let $A = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$, $B = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$, and $E = (A \cup B)_=$. Figure 4(b) depicts a congruence graph over E. The basic edges are shown in Figure 4(a); each corresponds to an equality in E. Since f is unary, each of the three derived edges has one parent path.

5.3   Colorable Congruence Graphs

Let A and B be sets of literals and let Σ_A and Σ_B be the sets of symbols that occur in A and B respectively. Terms, literals, and formulas over Σ_A will be called A-COLORABLE. Define B-COLORABLE analogously, and then define AB-COLORABLE to mean both A-colorable and B-colorable. COLORABLE entities are those that are either A-colorable or B-colorable.

Example 8. In Example 5, $\Sigma_A = \{x, z_1, z_2, z_3, \cdot\}$ and $\Sigma_B = \{y, z_1, z_2, z_3, \cdot\}$. Terms and equalities without occurrences of either x or y are AB-colorable. The term $x \cdot y$ and the equality $x \cdot z_2 = z_1 \cdot y$ are not colorable.

Extend the above definitions to edges of congruence graphs over $A \cup B$ so that an edge $\langle s, t \rangle$ has the same colorability attributes as the equality $s = t$. Finally, define a congruence graph to be COLORABLE if all its edges are colorable. Note that basic edges are always colorable.

Example 9. The congruence graphs derived from graphs in Figures 1–3 by removing their disequality edges are colorable. Half-filled vertices are AB-colorable, the dark ones are A-colorable but not B-colorable, and the light ones are B- but not A-colorable. If we add the derived edges $\langle x_i \cdot u_{i-1}, x_i \cdot v_{i-1} \rangle$ to the graph in Figure 2, they will be colorable; but if we add the derived edge $\langle x \cdot z_2, z_1 \cdot y \rangle$ to the graph in Figure 3, it will not be colorable.

Lemma 1. *If s and t are colorable terms and if $A, B \models s = t$, then there exists a colorable congruence graph over $(A \cup B)_=$ in which $s \sim t$.*

Proof. (Sketch) This is essentially Lemma 2 of [14], and the proof is constructive. Start with any congruence graph Γ with colorable vertices in which $s \sim t$ holds. If there are uncolorable edges, let $e = \langle f(u_1, \ldots, u_k), f(v_1, \ldots, v_k) \rangle$ be a minimal such edge in the derivation order. Thus, the parent paths $\overline{u_i v_i}$ are all

colorable, and each of them connects an A-colorable vertex with a B-colorable one. It follows that there exists an AB-colorable vertex w_i on each path $\overline{u_i v_i}$ (which may be one of its endpoints). The term $f(w_1, \ldots, w_k)$ is AB-colorable, so we can replace e in Γ with two edges $\langle f(u_1, \ldots, u_k), f(w_1, \ldots, w_k)\rangle$ and $\langle f(w_1, \ldots, w_k), f(v_1, \ldots, v_k)\rangle$, both of which are colorable. Now repeat the process until all uncolorable edges of Γ are eliminated.   □

5.4   Colored Congruence Graphs

Assume (without loss of generality) that the literal sets A, B are disjoint. A COLORING of a colorable congruence graph over $(A \cup B)_=$ is an assignment of a unique color A or B to each edge of the graph, such that

- basic edges are assigned the color of the set they belong to
- every edge colored X has both endpoints X-colorable ($X \in \{A, B\}$)

Thus, to color a colorable congruence graph, the only choice we have is with AB-colorable derived edges, and each of them can be colored arbitrarily. In the terminology of the interpolation game of §3, this means choosing which prover derives an AB-equality in a situation when either of them could do it. In Figure 4(b,c) we have two colored congruence graphs. They differ only in the coloring of $\langle f(z_3), f(z_4)\rangle$—the only derived edge with AB-colorable endpoints.
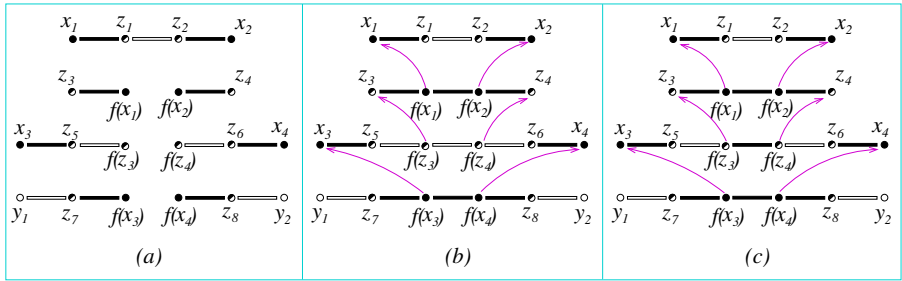
Fig. 4. Congruence graphs over $(A \cup B)_=$, with A and B from Example 7. The connection between a derived edge and its parent is indicated by a pair of arrows.

In a colored graph, we can speak of A-PATHS (whose edges are all colored A), and B-PATHS. There is also a color-induced factorization of arbitrary paths, where a FACTOR of a path is a maximal subpath consisting of equally colored edges. Clearly, every path can be uniquely represented as a concatenation of its factors, the consecutive factors having distinct colors.

5.5   The Interpolation Algorithm

A path $\overline{uv}$ in a congruence graph represents the equality $u = v$ between its endpoints. We will write $[\![\pi]\!]$ to denote the equality represented by the path π. More generally, if P is a set of paths, $[\![P]\!]$ is the corresponding set of equalities.

For every path π in a colored congruence graph, define the associated B-PREMISE SET $\mathcal{B}(\pi)$, the A-JUSTIFICATION $J(\pi)$, and the INTERPOLANT $I(\pi)$:

$$\mathcal{B}(\pi) = \begin{cases} \bigcup\{\mathcal{B}(\sigma) \mid \sigma \text{ is a factor of } \pi\} & \text{if } \pi \text{ has } \geq 2 \text{ factors} \\ \{\pi\} & \text{if } \pi \text{ is a } B\text{-path} \\ \bigcup\{\mathcal{B}(\sigma) \mid \sigma \text{ is a parent of an edge of } \pi\} & \text{if } \pi \text{ is an } A\text{-path} \end{cases} \quad (1)$$

$$J(\pi) = (\bigwedge[\![\mathcal{B}(\pi)]\!]) \Rightarrow [\![\pi]\!]$$

$$I(\pi) = \begin{cases} \bigwedge\{I(\sigma) \mid \sigma \text{ is a factor of } \pi\} & \text{if } \pi \text{ has } \geq 2 \text{ factors} \\ \bigwedge\{I(\sigma) \mid \sigma \text{ is a parent of an edge of } \pi\} & \text{if } \pi \text{ is a } B\text{-path} \\ J(\pi) \wedge \bigwedge\{I(\sigma) \mid \sigma \in \mathcal{B}(\pi)\} & \text{if } \pi \text{ is an } A\text{-path} \end{cases}$$

Empty parent paths σ in the definitions of $\mathcal{B}(\pi)$ and $I(\pi)$ can be ignored because $[\![\sigma]\!] = J(\sigma) = I(\sigma) = \mathsf{true}$ when σ is empty.

We also need a modified interpolant function I', expressed in terms of I as follows. The argument path π is first decomposed as $\pi = \pi_1 \theta \pi_2$, where θ is the largest subpath with B-colorable endpoints, or an empty path if there are no B-colorable vertices on π; then we define

$$I'(\pi) = I(\theta) \wedge \bigwedge\{I(\tau) \mid \tau \in \mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)\} \wedge \left(\bigwedge[\![\mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)]\!] \Rightarrow \neg[\![\theta]\!]\right)$$

This is well defined because π_1, θ, π_2 are uniquely determined by π if θ is not empty, and if θ is empty, the way we write π as $\pi_1 \pi_2$ is irrelevant. Note that when $\pi = \theta$, we have $I'(\pi) = I(\pi) \wedge \neg[\![\pi]\!]$.

The *EUF* GROUND INTERPOLATION ALGORITHM, given as inputs jointly inconsistent (disjoint) sets A, B of literals, proceeds as follows.

(**i1**) Run the congruence closure algorithm to find a congruence graph Γ over $(A \cup B)_=$ and a disequality $(s \neq t) \in A \cup B$ such that $s \sim t$ in Γ [§§5.1,5.2].

(**i2**) Modify Γ if necessary so that it is colorable [§5.3], then color it [§5.4].

(**i3**) If $(s \neq t) \in B$, return $I(\overline{st})$; if $(s \neq t) \in A$, return $I'(\overline{st})$.

Example 10. Let us run the algorithm for A, B in Example 7, using the colored congruence graph in Figure 4(b). Since $y_1 \neq y_2 \in B$, the interpolant is $I(\overline{y_1 y_2}) = I(\overline{y_1 z_7}) \wedge I(\overline{z_7 z_8}) \wedge I(\overline{z_8 y_2})$, and the first and third conjuncts are true. Thus, $I(\overline{y_1 y_2}) = I(\overline{z_7 z_8}) = J(\overline{z_7 z_8}) \wedge \bigwedge\{I(\sigma) \mid \sigma \in \mathcal{B}(\overline{z_7 z_8})\}$, so we need to compute $\mathcal{B}(\overline{z_7 z_8})$. We have $\mathcal{B}(\overline{z_7 z_8}) = \mathcal{B}(\overline{x_3 x_4}) = \mathcal{B}(\overline{x_3 z_5}) \cup \mathcal{B}(\overline{z_5 z_6}) \cup \mathcal{B}(\overline{z_6 x_4}) = \varnothing \cup \{\overline{z_5 z_6}\} \cup \varnothing = \{\overline{z_5 z_6}\}$. Thus, $J(\overline{z_7 z_8}) = (z_5 = z_6 \Rightarrow z_7 = z_8)$, which we denote ϕ_1. Continue the main computation: $I(\overline{y_1 y_2}) = \phi_1 \wedge I(\overline{z_5 z_6}) = \phi_1 \wedge I(\overline{z_3 z_4}) = J(\overline{z_3 z_4}) \wedge \bigwedge\{I(\sigma) \mid \sigma \in \mathcal{B}(\overline{z_3 z_4})\}$. Now, $\mathcal{B}(\overline{z_3 z_4}) = \mathcal{B}(\overline{x_1 x_2}) = \mathcal{B}(\overline{x_1 z_1}) \cup \mathcal{B}(\overline{z_1 z_2}) \cup \mathcal{B}(\overline{z_2 x_2}) = \varnothing \cup \{\overline{z_1 z_2}\} \cup \varnothing = \{\overline{z_1 z_2}\}$. Thus, $J(\overline{z_3 z_4}) = (z_1 = z_2 \Rightarrow z_3 = z_4)$, which we denote ϕ_2. Back to the main computation, $I(\overline{y_1 y_2}) = \phi_1 \wedge \phi_2 \wedge I(\overline{z_1 z_2}) = \phi_1 \wedge \phi_2 \wedge \mathsf{true} = \phi_1 \wedge \phi_2$. *Exercise:* Using the graph in Figure 4(c) results in a different interpolant: $I(\overline{y_1 y_2}) = (z_5 = f(z_3) \wedge z_6 = f(z_4) \wedge z_1 = z_2 \Rightarrow z_7 = z_8)$.

5.6   Correctness

Theorem 2. *With any jointly inconsistent sets A,B of EUF literals as inputs, the EUF ground interpolation algorithm (§5.5) terminates, returning an interpolant for A,B that is a conjunction of Horn clauses.*

Termination of our recursive definitions and other inductive arguments are proved using a well-founded relation $\prec$ over paths. Define $\sigma \prec' \pi$ to hold when one of the following holds:

- π has more than one factor, and σ is one of them
- σ is a parent path of an edge of π

Define $\prec$ as the transitive closure of $\prec'$. It is not difficult to see that the relation $\prec$ is well-founded. Note that minimal elements under $\prec$ are the paths all of whose edges are basic and of the same color.

The following equations redefine the set $\mathcal{B}(\pi)$ of B-premises and introduce the analogous set $\mathcal{A}(\pi)$ of A-PREMISES.

$$\mathcal{A}(\pi) = \{A\text{-factors of } \pi\} \cup \mathcal{A}(\{\text{parent paths of } B\text{-edges of } \pi\}) \qquad (2)$$
$$\mathcal{B}(\pi) = \{B\text{-factors of } \pi\} \cup \mathcal{B}(\{\text{parent paths of } A\text{-edges of } \pi\}) \qquad (3)$$

Here and in the sequel, we use the convention $f(P) = \bigcup\{f(\sigma) \,|\, \sigma \in P\}$ for extending a set-valued function f defined on paths to a function defined on sets of paths. Observe that (3) is just a restatement of (1). Also, the arguments in the recursive calls are smaller than π under the relation $\prec$, so termination is guaranteed. The basic properties of $\mathcal{A}$ are collected in the following lemma. The analogous properties of $\mathcal{B}$ follow by symmetry.

Lemma 2. *Let π be an arbitrary non-empty path in a congruence graph Γ.*

(i) If π is an A-path, then $\mathcal{A}(\pi) = \{\pi\}$; otherwise, $\sigma \prec \pi$ for every $\sigma \in \mathcal{A}(\pi)$.
(ii) If $\sigma \in \mathcal{A}(\pi)$, then $\mathcal{A}(\sigma) \subseteq \mathcal{A}(\pi)$.
(iii) If the endpoints of π are B-colorable, then the endpoints of all paths in $\mathcal{A}(\pi)$ are AB-colorable.

Proof. All three parts are proved by well-founded induction.

(i) If π is an A-colored path, then π is the only element of $\mathcal{A}(\pi)$ (by definition). If π is not an A-colored path and τ is an element of $\mathcal{A}(\pi)$, then τ is either an A-factor of π and so $\tau \prec \pi$ holds, or $\tau \in \mathcal{A}(\sigma)$ for some parent σ of a B-edge of π. In the latter case, $\tau \prec \pi$ holds because of $\sigma \prec \pi$ and the consequence $\tau \preceq \sigma$ of the induction hypothesis.

(ii) If σ is an A-factor of π, then $\mathcal{A}(\sigma) = \{\sigma\} \subseteq \mathcal{A}(\pi)$. If $\sigma \in \mathcal{A}(\tau)$ where τ is a parent path of a B-edge of π, then $\mathcal{A}(\sigma) \subseteq \mathcal{A}(\tau) \subseteq \mathcal{A}(\pi)$, the first inclusion by induction hypohesis, the second from the definition of $\mathcal{A}$.

(iii) Since parent paths of any B-edge must have B-colorable endpoints, for the inductive argument we only need to check that every A-factor of a path π with B-colorable endpoints has AB-colorable endpoints. Indeed, A-colorability of endpoints of A-factors is obvious. For B-colorability, observe that an endpoint of an A-factor of π is either also an endpoint of a B-factor of π, or an endpoint of π itself. $\qquad\square$

The following lemma justifies the names *A-premises* and *B-premises*. In accordance with the notation of §3, *B*-premises are the *B*-paths whose summaries, if added to *A*, make the derivation of $[\![\pi]\!]$ possible.

Lemma 3. $A, [\![\mathcal{B}(\pi)]\!] \models [\![\pi]\!]$ *and* $B, [\![\mathcal{A}(\pi)]\!] \models [\![\pi]\!]$, *for every path π in Γ.*

Proof. We prove the first claim only, by well-founded induction based on $\prec$. Viewing π as the concatenation of its *B*-factors and *A*-edges, we have by transitivity

$$[\![B\text{-factors of } \pi]\!],\ [\![A\text{-edges of } \pi]\!] \models [\![\pi]\!]$$

and then, since $A \models [\![e]\!]$ for every basic *A*-edge e (by definition of edge coloring),

$$A,\ [\![B\text{-factors of } \pi]\!],\ [\![\text{derived } A\text{-edges of } \pi]\!] \models [\![\pi]\!].$$

For every derived edge e we have $[\![\text{parents of } e]\!] \models [\![e]\!]$. Thus,

$$A,\ [\![B\text{-factors of } \pi]\!],\ [\![\text{parents of } A\text{-edges of } \pi]\!] \models [\![\pi]\!],$$

so it suffices to prove $A, [\![\mathcal{B}(\pi)]\!] \models [\![\sigma]\!]$ for every σ that is either a *B*-factor of π or a parent of an *A*-edge of π. In the first case, the claim clearly holds since $\sigma \in \mathcal{B}(\pi)$. In the second case, we have $\sigma \prec \pi$, so the induction hypothesis gives us $A, [\![\mathcal{B}(\sigma)]\!] \models [\![\sigma]\!]$. To finish the proof, just use $\mathcal{B}(\sigma) \subseteq \mathcal{B}(\pi)$, which is true by Lemma 2(ii). $\qquad\square$

Define the CUMULATIVE SET OF PREMISES (cf. §3)

$$\mathcal{P}(\pi) \;=\; \{\pi\} \cup \mathcal{P}(\mathcal{B}(\mathcal{A}(\pi))) \tag{4}$$

Termination of this recursive definition follows from Lemma 2(i).

Lemma 4. $I(\pi) \;=\; \bigwedge \{J(\sigma) \,|\, \sigma \in \mathcal{A}(\mathcal{P}(\pi))\}.$

Proof. Let $\mathcal{P}'(\pi) = \mathcal{A}(\mathcal{P}(\pi))$. From (4), we have

$$\mathcal{P}'(\pi) \;=\; \mathcal{A}(\pi) \cup \mathcal{P}'(\mathcal{B}(\mathcal{A}(\pi))) \tag{5}$$

It suffices to check that

$$\mathcal{P}'(\pi) = \begin{cases} \bigcup\{\mathcal{P}'(\sigma) \,|\, \sigma \text{ is a factor of } \pi\} & \text{if } \pi \text{ has } \geq 2 \text{ factors} \\ \bigcup\{\mathcal{P}'(\sigma) \,|\, \sigma \text{ is a parent of an edge of } \pi\} & \text{if } \pi \text{ is a } B\text{-path} \\ \{\pi\} \cup \bigcup\{\mathcal{P}'(\sigma) \,|\, \sigma \in \mathcal{B}(\pi)\} & \text{if } \pi \text{ is an } A\text{-path} \end{cases}$$

For the first case, suppose $\pi = \pi_1 \cdots \pi_k$ is the factorization of π. By definition of $\mathcal{A}$, we have $\mathcal{A}(\pi) = \mathcal{A}(\pi_1) \cup \cdots \cup \mathcal{A}(\pi_k)$. The desired equation $\mathcal{P}'(\pi) = \mathcal{P}'(\pi_1) \cup \cdots \cup \mathcal{P}'(\pi_k)$ then follows from (5). Assume now $\pi = e_1 \cdots e_k$ is a *B*-path. By definition of $\mathcal{A}$, we have $\mathcal{A}(\pi) = \mathcal{A}(E_1) \cup \cdots \cup \mathcal{A}(E_k)$, where E_i is the set of parent paths of the edge e_i. Again, the desired equation $\mathcal{P}'(\pi) = \mathcal{P}'(E_1) \cup \cdots \cup \mathcal{P}'(E_k)$ follows from (5). Finally, assume that π is an *A*-path. Now $\mathcal{A}(\pi) = \{\pi\}$ and so $\mathcal{P}'(\pi) = \{\pi\} \cup \mathcal{P}'(\mathcal{B}(\pi))$, again by (5). $\qquad\square$

Lemma 5. $B, I(\pi) \models [\![\pi]\!]$, *for every path π in Γ with B-colorable endpoints.*

Proof. We argue by induction along $\prec$. Let σ be an arbitrary A-premise of π and τ an arbitrary B-premise of σ. The endpoints of τ are B-colorable, because in general, every B-premise of any path is a B-factor of some path, and every B-factor of any path has B-colorable endpoints. Thus, the induction hypothesis applies to τ and we have $B, I(\tau) \models [\![\tau]\!]$. From equation (5) we have $\mathcal{P}'(\tau) \subseteq \mathcal{P}'(\pi)$, so we can derive $I(\pi) \models I(\tau)$ using Lemma 4. Thus, $B, I(\pi) \models [\![\tau]\!]$ for every $\tau \in \mathcal{B}(\sigma)$. By Lemma 4, $I(\pi)$ contains $J(\sigma)$ as a conjunct; therefore, $B, I(\pi) \models [\![\sigma]\!]$. Since σ here is an arbitrary element of $\mathcal{A}(\pi)$, the second claim of Lemma 3 finishes the proof.   □

Proof of Theorem 2. The algorithm terminates because all pertinent functions have been proven terminating.

Let $s \neq t$ be the critical disequality obtained in the step (**i1**) of the algorithm. Let π be the path $\overline{st}$, and let $\overline{st} = \pi_1 \theta \pi_2$, as in the definition of $I'(\pi)$. The two cases to consider, $s \neq t \in B$ and $s \neq t \in A$, will be referred to as Cases 1 and 2 respectively. Let ϕ be the returned formula—$I(\pi)$ in Case 1; $I'(\pi)$ in Case 2.

(i) ϕ is an AB-colorable conjunction of Horn clauses. For any σ with AB-colorable endpoints, $J(\sigma)$ is an AB-colorable Horn clause. If π has B-colorable endpoints, then so do all paths in $\mathcal{P}(\pi)$ and so, by Lemma 2(iii), all paths in $\mathcal{A}(\mathcal{P}(\pi))$ have AB-colorable endpoints. With Lemma 4, this proves Case 1. For Case 2, observe that if θ is empty, then $I(\theta) = [\![\theta]\!] = \mathsf{true}$; otherwise, θ has AB-colorable endpoints. Also, π_1 and π_2 are A-paths, so by the dual of Lemma 2(iii), all paths in $\mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)$ have AB-colorable endpoints. These facts suffice to derive the proof of Case 2 from the already proved Case 1.

(ii) $A \models \phi$. By the first claim of Lemma 3, $A \models J(\sigma)$ holds for every σ. This suffices for Case 1. For Case 2 then, we only need to check that the last conjunct of $I'(\pi)$ is implied by A, which amounts to showing $A, [\![\mathcal{B}(\pi_1)]\!], [\![\mathcal{B}(\pi_2)]\!] \models \neg[\![\theta]\!]$. This indeed follows from the first claim of Lemma 3, the transitivity entailment $[\![\pi_1]\!], [\![\theta]\!], [\![\pi_2]\!] \models [\![\pi]\!]$, and the assumption $\neg[\![\pi]\!] \in A$.

(iii) $B, \phi \models \mathsf{false}$. In Case 1, we have $\neg[\![\pi]\!] \in B$, so Lemma 5 finishes the proof. In Case 2, Lemma 5 implies $B, I'(\pi) \models [\![\theta]\!]$ and $B, I'(\pi) \models I(\tau)$ for every $\tau \in \mathcal{B}(\pi_1) \cup \mathcal{B}(\pi_2)$. These consequences of $B \cup I'(\pi)$ contradict the last conjunct of $I'(\pi)$.   □

6   Comparison with McMillan's Algorithm

Our *EUF* ground interpolation algorithm is, as far as we know, the only alternative to McMillan's algorithm [8]. The latter constructs an interpolant for A, B from the proof of $A, B \models \mathsf{false}$ derived in a formal system ($\mathcal{E}$, say) with rules for introducing hypotheses (equalities from $A \cup B$), reflexivity, symmetry, transitivity, congruence, and contradiction (deriving false from an equality and its negation). The algorithm proceeds top down by annotating each intermediate derived equality $u = v$ (or false in the final step) with a quadruple of the

form $[u', v', \rho, \gamma]$, where u', v' are terms and ρ, γ are AB-colorable formulas. The annotation of each derived equality is obtained from annotations of the equalities occurring in the premises of the corresponding rule application. The exact computation of annotations is specified by 11 rules, each corresponding to a case (depending on colors of the terms involved) of one of the original six rules. An invariant that relates a derived intermediate equality with its annotation is formulated and all 11 rules are proved to preserve the invariant. The invariant implies that if $[u', v', \rho, \gamma]$ is the annotation of false, then $\rho \Rightarrow \gamma$ is an interpolant for A, B. It can be shown that ρ is always a conjunction of Horn clauses, and γ is a conjunction of equalities and at most one disequality.

There is a clear relationship between proofs in the formal system $\mathcal{E}$ and congruence graphs from which our interpolants are derived. The main difference is that in congruence graphs, paths condense inferences by reflexivity, symmetry, and transitivity. A congruence graph provides a big-step proof that, if necessary, can be expanded into a proof in the system $\mathcal{E}$.

In Example 2 (Figure 1(a)) our algorithm looks at the path $\overline{y_3 z_4}$, summarizes its only A-factor, producing the interpolant $z_1 = z_4$. McMillan's algorithm processes the path edge-by-edge, eagerly summarizing A-chains with AB-colorable endpoints, so that the interpolant it produces is $z_1 = z_2 \wedge z_2 = f(z_3) \wedge f(z_3) = z_4$.

For the second difference, consider Example 7 (Figure 4(b)) where McMillan's algorithm produces an entangled version $(z_1 = z_2 \wedge (z_3 = z_4 \Rightarrow z_5 = z_6)) \Rightarrow z_3 = z_4 \wedge z_7 = z_8$ of our interpolant $(z_1 = z_2 \Rightarrow z_3 = z_4) \wedge (z_5 = z_6 \Rightarrow z_7 = z_8)$, computed in Example 10. In general, McMillan's algorithm accumulates B-justifications (duals of our $J(\sigma)$) in the ρ-part of the annotation and keeps them past their one-time use to derive a particular conjunct of γ.

The third difference is in creating auxiliary AB-terms ("equality interpolants", in the terminology of [14]) to split derivations of equalities in which one side is not A-colorable and the other is not B-colorable, as in Example 5. We introduce an absolute minimum of such terms in the preliminary step (**i2**) of our algorithm, where these terms are added to make the congruence graph colorable. In contrast, McMillan's algorithm introduces these terms "on-the-fly", as in the example illustrated in Figure 5. When it derives the equality $x_1 = z_2$, its annotation is $[z_1, z_2, \mathsf{true}, \mathsf{true}]$, then when it uses the congruence rule to derive $f(x_1) = f(z_2)$, this equality gets annotated with $[f(z_1), f(z_2), \mathsf{true}, \mathsf{true}]$, and the term $f(z_1)$ becomes part of the final interpolant $z_3 = f(z_1) \wedge f(z_2) = z_4$. On the other hand, our algorithm recognizes the edge $\langle f(x_1), f(z_2) \rangle$ as A-colorable and does not split it; the interpolant it produces is $z_1 = z_2 \Rightarrow z_3 = z_4$.

The final difference is in flexibility. McMillan's algorithm is fully specified and leaves little room for variation. On the other hand, the actions in the step (**i2**) of our algorithm are largely non-deterministic. Our current implementation chooses to minimize the number of vertices in the colorable modification of Γ, and then colors the graph with a strategy that eagerly minimizes the number of factors in the relevant paths. Other choices are yet to be explored.

In general, our algorithm produces smaller and simpler interpolants. For experimental confirmation, we used the state-of-the-art implementation of
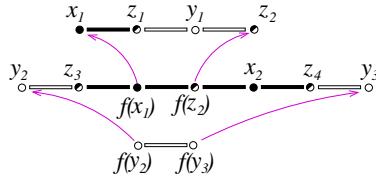
Fig. 5. A colored congruence graph for $A = \{x_1 = z_1, z_3 = f(x_1), f(z_2) = x_2, x_2 = z_4\}$ and $B = \{z_1 = y_1, y_1 = z_2, y_2 = z_3, z_4 = y_3, f(y_2) \neq f(y_3)\}$ with two derived edges $\langle f(x_1), f(z_2) \rangle$ and $\langle f(y_2), f(y_3) \rangle$.
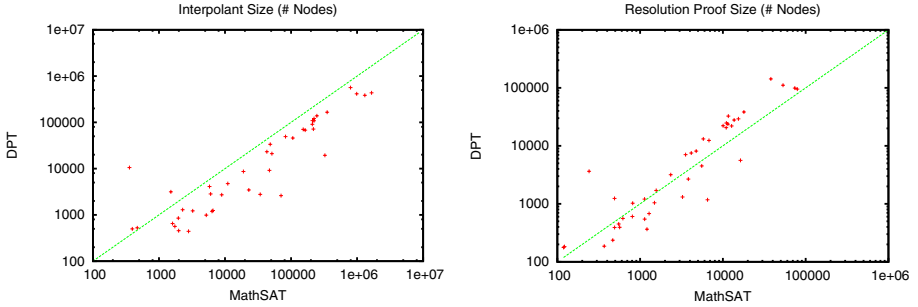

Fig. 6. *DPT vs. MathSAT* on 45 benchmarks from the *MathSAT* library derived by partitioning unsatisfiable SMT-LIB benchmarks [2].

McMillan's algorithm in *MathSAT* [3] and compared it against our interpolation-generating extension of the *DPT* solver [1]. Two other relevant components—the propositional interpolation algorithm, and the algorithm for combining propositional and theory interpolation in a DPLL($\mathcal{T}$) framework [8,3]—are the same in *MathSAT* and *DPT*, and therefore unlikely to substantially affect the comparison. The last factor to be accounted for in this comparison is the size of the resolution proofs derived from the DPLL search within each solver. These sizes being comparable, we can eliminate the differences in propositional reasoning as a cause for *DPT* producing smaller interpolants.

We ran both solvers on 45 *EUF* interpolation benchmarks selected from the set of 100 that are used in [3]. (In the remaining 55 benchmarks, either all formulas in A are B-colorable, or all formulas in B are A-colorable, so one of the formulas A, $\neg B$ is an easily obtained interpolant.) Both solvers computed 42 interpolants, timing out in $100s$ on the same three benchmarks. Runtimes were comparable, with *DPT* being slightly faster. Figure 6 shows the sizes of interpolants produced: *DPT* interpolants are, on average, 3.8 times smaller, in spite of *DPT* proofs being, on average, 1.7 times larger.

7   Conclusion

Our analysis of the interpolation for the theory of equality was motivated by the central role this theory plays in SMT solving, and by the practical applicability

of interpolant-producing SMT solvers. The algorithm we obtained is easy to implement on top of the standard congruence closure procedure. It generates interpolants of a simple logical form and smaller size than those produced by the alternative method.

We identified *congruence graphs* as a convenient structure to represent proofs in *EUF* and to derive interpolants. The possibilities for global analysis and transformations of these graphs go beyond what we have explored. Our algorithm provides a basis for further refinement and multiple implementations. This flexibility may prove useful when the notion of interpolant quality is better understood.

References

1. Decision Procedure Toolkit (2008), `http://www.sourceforge.net/projects/DPT`
2. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library, SMT-LIB (2008), `http://www.SMT-LIB.org`
3. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
4. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Journal of Symbolic Logic 22(3), 269–285 (1957)
5. Ghilardi, S.: Model-theoretic methods in combined constraint satisfiability. Journal of Automated Reasoning 33(3–4), 221–249 (2005)
6. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: Young, M., Devanbu, P.T. (eds.) SIGSOFT FSE, pp. 105–116. ACM, New York (2006)
7. McMillan, K.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
8. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science 345(1), 101–121 (2005)
9. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
10. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM 27(2), 356–364 (1980)
11. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005)
12. Pudlák, P.: Lower bounds for resolution and cutting planes proofs and monotone computations. Journal of Symbolic Logic 62(3) (1997)
13. Tinelli, C.: Cooperation of background reasoners in theory reasoning by residue sharing. Journal of Automated Reasoning 30(1), 1–31 (2003)
14. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Satisfiability Procedures for Combination of Theories Sharing Integer Offsets

Enrica Nicolini, Christophe Ringeissen, and Michaël Rusinowitch

LORIA & INRIA Nancy Grand Est, France
`FirstName.LastName@loria.fr`

Abstract. We present a novel technique to combine satisfiability procedures for theories that model some data-structures and that share the integer offsets. This procedure extends the Nelson-Oppen approach to a family of non-disjoint theories that have practical interest in verification. The result is derived by showing that the considered theories satisfy the hypotheses of a general result on non-disjoint combination. In particular, the capability of computing logical consequences over the shared signature is ensured in a non trivial way by devising a suitable complete superposition calculus.

1   Introduction

Satisfiability procedures for fragments of Arithmetics and data structures such as arrays and lists are at the core of many state-of-the-art verification tools, and their design and correct implementation is a hard task [5]. To overcome this difficulty, there is an obvious need for developing general and systematic methods to build decision procedures. Two important approaches have been investigated based respectively on combination and rewriting.

The *combination approach* for the satisfiability problem has been initiated in [14,17]. The methodology is to combine existing decision procedures for component theories in order to get a decision procedure for the union of the theories. In particular, the combination à la Nelson-Oppen is the core of many verification tools, even if the implementations often exploit ideas quite far from the original schema (see, e.g. [12,4]). This method assumes that component theories have disjoint signatures. An extension to the non-disjoint case has been proposed in [8,10], where the cooperation between the decision procedures relies on their capabilities of computing logical consequences built over the shared signature.

The *rewriting approach* allows us to flexibly build satisfiability procedures [2,1] based on a general calculus for automated deduction, namely the superposition calculus [16]. Hence, to obtain satisfiability procedures becomes easy by using an (almost) off-the-shelf theorem prover implementing superposition.

These two approaches are complementary for two main reasons. First, combination techniques allow us to incorporate theories which are difficult to handle using rewriting techniques, such as Linear Arithmetics. Second, rewriting techniques are of prime interest to design satisfiability procedures which can be efficiently plugged into the disjoint combination framework [11]. In some particular

cases, the rewriting approach is an alternative to the combination approach by allowing us to build superposition-based satisfiability procedures for combinations of finitely axiomatized theories, including the theory of Integer Offsets [1,3], but these theories must be over *disjoint* signatures.

In this paper, we show how to apply a superposition calculus to build decision procedures that can be plugged into the *non-disjoint* combination framework. We focus on theories sharing Integer Offsets. We present a superposition calculus dedicated to this theory and show the soundness of this new calculus for several *non-disjoint* extensions of this theory. The interest of combining counter arithmetic and uninterpreted functions in verification is advocated in [6], where uninterpreted functions are used for abstracting data and Integer Offsets allows us to express counters and a form of pointers, thanks to the successor function s and 0. For instance, the possibility of using Integer Offsets enables us to consider (and combine) several models of lists:

- We can use the classical model of lists à la LISP, using $\mathsf{cons}, \mathsf{car}, \mathsf{cdr}$ operators, augmented with a length function ℓ defined as follows: $\ell(\mathsf{cons}(e, x)) = \mathsf{s}(\ell(x))$ and $\ell(\mathsf{nil}) = 0$. In general, lists are over arbitrary elements but we may use also lists over integer elements.
- We can consider lists defined as records with two fields, the first one for the list itself, and the second one to store its length. Let us consider the operator $\mathsf{rselect_i}$ to access to the i-th field of a record, $rcons(e, r)$ denotes the record obtained by adding an element e to the list of r, and $rnil$ denotes the record corresponding to the empty list, we have the following axiomatization:

$$\mathsf{rselect_1}(rcons(e, r)) = \mathsf{cons}(e, \mathsf{rselect_1}(r)) \qquad \mathsf{rselect_1}(rnil) = nil$$
$$\mathsf{rselect_2}(rcons(e, r)) = \mathsf{s}(\mathsf{rselect_2}(r)) \qquad \mathsf{rselect_2}(rnil) = 0$$

This model of lists can be seen as a refinement of the first model in which one has a direct access to its "cardinality".

The combination framework presented in the paper can be applied to decide the satisfiability of ground formulas expressed in the union of these two models of lists (provided both models use distinct names for list operators). Roughly speaking, such combination is useful to verify for instance that two programs written using different models of lists are "equivalent".

Plan of the paper. After this introduction, Section 2 gives the main concepts and notations related to first-order theories. Section 3 presents the non-disjoint combination framework. In Section 4, we present a superposition calculus dedicated to the theory of Integer Offsets. In Section 5, we give some examples of theories for which this superposition calculus can be turned into decision procedures. In Section 6, we show that this superposition calculus can be also applied to deduce logical shared consequences. Moreover, all the requirements for applying the non-disjoint combination framework are satisfied by the extensions of Integer Offsets we are interested in. Finally, Section 7 concludes with some final remarks and a description of future work. For lack of space, proofs are omitted and can be found in [15].

2   Preliminaries

A *signature* Σ is a set of functions and predicate symbols (each endowed with the corresponding arity). We assume the binary equality predicate symbol '=' to be always present in any signature Σ (so, if $\Sigma = \emptyset$, then Σ does not contain other symbols than equality). The signature obtained from Σ by adding a set $\underline{a}$ of new constants (i.e., 0-ary function symbols) is denoted by $\Sigma^{\underline{a}}$. Σ-*atoms*, Σ-*literals*, Σ-*clauses*, and Σ-*formulae* are defined in the usual way. A set of Σ-literals is called a Σ-*constraint*. Terms, literals, clauses and formulae are called *ground* whenever no variable appears in them; *sentences* are formulae in which free variables do not occur. Given a function symbol f, a f-rooted term is a term whose top-symbol is f.

From the semantic side, we have the standard notion of a Σ-*structure* $\mathcal{M} = (M, \mathcal{I})$: this is a support set M endowed with an arity-matching interpretation $\mathcal{I}$ of the function and predicate symbols from Σ. Truth of a Σ-formula in $\mathcal{M}$ is defined in any one of the standard ways. If $\Sigma_0 \subseteq \Sigma$ is a subsignature of Σ and if $\mathcal{M}$ is a Σ-structure, the Σ_0-*reduct* of $\mathcal{M}$ is the Σ_0-structure $\mathcal{M}_{|\Sigma_0}$ obtained from $\mathcal{M}$ by forgetting the interpretation of function and predicate symbols from $\Sigma \setminus \Sigma_0$.

A collection of Σ-sentences is a Σ-theory, and a Σ-theory T admits *quantifier elimination* iff for every formula $\varphi(\underline{x})$ there is a quantifier-free formula (over the same free variables $\underline{x}$) $\varphi'(\underline{x})$ such that $T \models \varphi(\underline{x}) \Leftrightarrow \varphi'(\underline{x})$.

In this paper, we are concerned with the *(constraint) satisfiability problem* for a theory T, also called the T-satisfiability problem, which is the problem of deciding whether a Σ-constraint is satisfiable in a model of T (and, if so, we say that the constraint is T-satisfiable). Notice that a constraint may contain variables: since these variables may be equivalently replaced by free constants, we can reformulate the constraint satisfiability problem as the problem of deciding whether a finite conjunction of ground literals in a simply expanded signature $\Sigma^{\underline{a}}$ is true in a $\Sigma^{\underline{a}}$-structure whose Σ-reduct is a model of T.

3   Non-disjoint Combination of Theories

We are interested in applying a general method for the combination of satisfiability procedures in unions of non-disjoint theories. This method extends the Nelson-Oppen combination method known for unions of signature-disjoint theories, and leads to the following result:

Theorem 1. *[10] Consider two theories T_1, T_2 in signatures Σ_1, Σ_2 and suppose that:*

1. *both T_1, T_2 have decidable constraint satisfiability problem;*
2. *there is some theory T_0 in the signature $\Sigma_1 \cap \Sigma_2$ such that:*
   - *T_0 is universal;*
   - *T_1, T_2 are both T_0-compatible;*
   - *T_0 is Noetherian;*
   - *T_1, T_2 are both effectively Noetherian extensions of T_0.*

Then the $(\Sigma_1 \cup \Sigma_2)$-theory $T_1 \cup T_2$ also has decidable constraint satisfiability problem.

The decidability result of Theorem 1 is obtained by relying on the available decision procedures for T_1 and T_2, and cooperating them through an exchange of information over the shared signature $\Sigma_1 \cup \Sigma_2$. There are three crucial points in this schema: first of all, one should identify conditions sufficient to guarantee the correctness of the resulting procedure: this has been addressed requiring that the component theories must be both compatible with a common sub-theory. Secondly, one should ensure the capability of computing the information to be exchanged: this issue is encoded into the requirement that the two theories T_1 and T_2 are "effectively Noetherian extensions" of a common subtheory T_0. Finally, one should guarantee that the exchange process eventually halts: the termination of the whole procedure is ensured thanks to the so called Noetherianity of T_0. Let us explain in more details what the aforementioned requirements are.

Definition 1 (T_0-compatibility). *Let T be a theory in the signature Σ and let T_0 be a universal theory in a subsignature $\Sigma_0 \subseteq \Sigma$. We say that T is T_0-compatible iff $T_0 \subseteq T$ and there is a Σ_0-theory $T_0^\star$ such that*

- *(i) $T_0 \subseteq T_0^\star$;*
- *(ii) $T_0^\star$ has quantifier elimination;*
- *(iii) every Σ_0-constraint which is satisfiable in a model of T_0 is satisfiable also in a model of $T_0^\star$;*
- *(iv) every Σ-constraint which is satisfiable in a model of T is satisfiable also in a model of $T_0^\star \cup T$.*

The requirements (i) to (iii) make the theory $T_0^\star$ unique, provided it exists ($T_0^\star$ is the so-called *model completion* of T_0). These requirements are a generalization of the stable infiniteness requirement of the Nelson-Oppen combination procedure: in fact, if T_0 is the empty theory in the empty signature, $T_0^\star$ is the theory axiomatizing an infinite domain, so that (iii) holds trivially and (iv) is precisely stable infiniteness.

Example 1. Let us consider the theory of Integer Offsets T_I:

$\boxed{T_I}$ rules the behaviour of the successor function s and the constant 0. T_I has the mono-sorted signature $\Sigma_I := \{0 : \text{INT}, \mathsf{s} : \text{INT} \to \text{INT}\}$, and it is axiomatized as follows:

$\forall x\ \mathsf{s}(x) \neq 0$
$\forall x, y\ \mathsf{s}(x) = \mathsf{s}(y) \Rightarrow x = y$
$\forall x\ x \neq t(x)$ for all the terms $t(x)$ over Σ_I that properly contain x

T_I is a universal theory that admits model completion: indeed, if we add to T_I the axiom $\forall x(x \neq 0 \Rightarrow \exists y\ x = \mathsf{s}(y))$, we obtain a theory $T_I^\star$ that admits quantifier elimination (see, e.g. [7]) and such that every constraint that is satisfiable in a model of T_I is satisfiable also in a model of $T_I^\star$. To justify the last claim, it is

sufficient to observe that each model of T_I can be extended to a model of T_I simply by adding recursively to each element different from (the interpretation of) 0 a "predecessor". Since this operation does not affect the truth of any constraint, we obtain that the condition (iii) is satisfied.

Now, for any theory $T \supseteq T_I$ over a signature $\Sigma \supseteq \Sigma_I$ the T_I-compatibility requirement simply reduces to the following condition: every constraint Γ that is satisfiable in a model of T must be satisfiable also in a model of $T \cup \forall x (x \neq 0 \Rightarrow \exists y \, x = \mathsf{s}(y))$.

The method for cooperating the satisfiability procedures makes use of the capability of deducing logical consequences over the shared signature. In order to ensure the termination when deducing those logical consequences, we rely on Noetherian theories. Intuitively, a theory is Noetherian if there exists only a finite number of atoms that are not redundant when reasoning modulo T_0.

Definition 2 (Noetherian Theory). *A Σ_0-theory T_0 is* Noetherian *if and only if for every* finite *set of free constants $\underline{a}$, every infinite ascending chain*

$$\Theta_1 \subseteq \Theta_2 \subseteq \cdots \subseteq \Theta_n \subseteq \cdots$$

of sets of ground $\Sigma_0^{\underline{a}}$-atoms is eventually constant modulo T_0, i.e. there is an n such that $T_0 \cup \Theta_n \models A$, for every natural number m and atom $A \in \Theta_m$.

Example 2. (Example 1 continued). Many examples of Noetherian theories come from the formalization of algebraic structures, but an interesting class of Noetherian theories consists in all the theories whose signature contains only constants and one unary function symbol [9,18]. Thus, the theory of Integer Offsets T_I enjoys this property.

Let us consider now a theory $T \supseteq T_0$ with signatures $\Sigma \supseteq \Sigma_0$, and suppose we want to discover, given an arbitrary set of ground clauses Θ over Σ, a "complete set" of logical positive consequences of Θ over Σ_0, formalized by the notion of T_0-*basis*.

Definition 3 (T_0-basis). *Given a finite set Θ of ground clauses (built out of symbols from Σ and possibly further free constants) and a finite set of free constants $\underline{a}$, a T_0-basis for Θ w.r.t. $\underline{a}$ is a set Δ of positive ground $\Sigma_0^{\underline{a}}$-clauses such that*

(i) $T \cup \Theta \models C$, for all $C \in \Delta$ and
(ii) if $T \cup \Theta \models C$ then $T_0 \cup \Delta \models C$, for every positive ground $\Sigma_0^{\underline{a}}$-clause C.

Notice that in the definition of a basis we are interested only in positive ground clauses: the exchange of positive information is sufficient to ensure the completeness of the resulting procedure. The interest in Noetherian theories lies in the fact that, for every set of Σ-clauses Θ and for every set $\underline{a}$ of constants, a finite T_0-basis for Θ w.r.t. $\underline{a}$ always exists. Unfortunately, a basis for a Noetherian theory needs not to be computable; this motivates the following definition corresponding to the last hypothesis of Theorem 1.

Definition 4. *Given a finite set $\underline{a}$ of free constants, a T-residue enumerator for T_0 w.r.t. $\underline{a}$ is a computable function $Res_T^a(\Gamma)$ mapping a set of Σ-clauses Γ to a finite T_0-basis for Γ w.r.t. $\underline{a}$*[1]*. A theory T is an* effectively Noetherian extension *of T_0 if and only if T_0 is Noetherian and there exists a T-residue enumerator for T_0 w.r.t. every finite set $\underline{a}$ of free constants.*

Now we are ready to give a more detailed picture of the procedure that is the core of Theorem 1, and that extends the Nelson-Oppen combination method to theories over non disjoint signatures.

Algorithm 1. Extending Nelson-Oppen

Step 1. Purify the finite **input** set of ground literals Γ, thus producing a finite set Γ_1 of ground Σ_1^a-literals and finite set Γ_2 of ground Σ_2^a-literals s.t. $\Gamma_1 \cup \Gamma_2$ is $T_1 \cup T_2$-equisatisfiable with Γ.

Step 2. Using the T_i-residue enumerator Res_{T_i}, check the output of $Res_{T_i}(\Gamma_i)$:
  If $Res_{T_i}(\Gamma_i) = \Delta_i$ and $\Delta_i \neq \bot$ for each $i \in \{1, 2\}$, then
    Step 2.1. For each $D \in \Delta_i$ such that $T_j \cup \Gamma_j \not\models D$, $(i \neq j)$, add D to Γ_j
    Step 2.2. If Γ_1 or Γ_2 has been changed in **Step 2.1**, then rerun **Step 2**
  Else **return** *"unsatisfiable"*

Step 3. If this step is reached, **return** *"satisfiable"*.

In the following we will show how to discover theories that are amenable to be combined via the above schema and that share the theory of Integer Offsets. More in detail, we will focus on a particular extension of the superposition calculus that will proved to be a decision procedure for theories extending T_I and that will provide residue enumerators for T_I.

4   Superposition Calculus for Integer Offsets

Recent literature has focused on the possibility of using the superposition calculus in order to decide the satisfiability of ground formulae modulo the theory of Integer Offsets and some disjoint extensions [1,3]. Contrary to those papers, we are interested in a superposition-based calculus to deal with non-disjoint extensions of Integer Offsets, being able to constraint the successor symbol with additional axioms.

Let us consider the axiomatization of the theory of Integer Offsets T_I defined in Example 1. Our aim is to develop a calculus able to take into account the axioms of T_I into a framework based on superposition. To this aim, let us consider a presentation of the superposition calculus specialized for reasoning over sets of literals, whose rules are described in Figures 1 and 2, augmented with the four more rules over ground terms presented in Figure 3.

Let us adapt the standard definition of *derivation* to the calculus we are interested in:

---

[1] If Γ is T-unsatisfiable, then without loss of generality a residue enumerator can always return the singleton set containing the empty clause.

$$\text{Superposition} \quad \frac{l[u'] = r \quad u = t}{(l[t] = r)\sigma} \quad (i), (ii)$$

$$\text{Paramodulation} \quad \frac{l[u'] \neq r \quad u = t}{(l[t] \neq r)\sigma} \quad (i), (ii)$$

$$\text{Reflection} \quad \frac{u' \neq u}{\perp}$$

where σ is the most general unifier of u and u', u' is not a variable in *Superposition* and *Paramodulation*, L is a literal, $\perp$ is the syntactic sign used to denote the inconsistency and the following hold: *(i)* $u\sigma \not\preceq t\sigma$, *(ii)* $l[u']\sigma \not\preceq r\sigma$.

Fig. 1. Expansion Inference Rules

$$\text{Subsumption} \quad \frac{S \cup \{L, L'\}}{S \cup \{L\}} \quad \text{if } L\vartheta \equiv L' \text{ for some substitution } \vartheta$$

$$\text{Simplification} \quad \frac{S \cup \{L[l'], l = r\}}{S \cup \{L[r\vartheta], l = r\}} \quad \begin{array}{l} \text{if } l' \equiv l\vartheta, \; r\vartheta \prec l\vartheta, \text{ and} \\ (l\vartheta = r\vartheta) \prec L[l\vartheta] \end{array}$$

$$\text{Deletion} \quad \frac{S \cup \{t = t\}}{S}$$

where L and L' are literals and S is a set of literals.

Fig. 2. Contraction Inference Rules

$$R1 \quad \frac{S \cup \{\mathsf{s}(u) = \mathsf{s}(v)\}}{S \cup \{u = v\}} \quad \text{if } u \text{ and } v \text{ are ground terms}$$

$$R2 \quad \frac{S \cup \{\mathsf{s}(u) = t, \mathsf{s}(v) = t\}}{S \cup \{\mathsf{s}(v) = t, u = v\}} \quad \begin{array}{l} \text{if } u, v \text{ and } t \text{ are ground terms and} \\ \mathsf{s}(u) \succ t, \; \mathsf{s}(v) \succ t \text{ and } u \succ v \end{array}$$

$$C1 \quad \frac{S \cup \{\mathsf{s}(t) = 0\}}{S \cup \{\mathsf{s}(t) = 0\} \cup \perp} \quad \text{if } t \text{ is a ground term}$$

$$C2 \quad \frac{S \cup \{\mathsf{s}^n(t) = t\}}{S \cup \{\mathsf{s}^n(t) = t\} \cup \perp} \quad \text{if } t \text{ is a ground term and } n \in \mathbb{N}$$

where S is a set of literals and $\perp$ is the symbol for the inconsistency.

Fig. 3. Ground reduction Inference Rules

Definition 5. *Let $\mathcal{SP}_I$ be the calculus depicted in Figures 1, 2 and 3. A derivation (δ) with respect to $\mathcal{SP}_I$ is a (finite or infinite) sequence of sets of literals $S_1, S_2, S_3, \ldots, S_i, \ldots$ such that, for every i, it happens that:*

(i) S_{i+1} is obtained from S_i adding a literal obtained by the application of one of the rules in Figures 1, 2 and 3 to some literals in S_i;

(ii) S_{i+1} is obtained from S_i removing a literal according to one of the rules in Figures 2 or to the rule R1 or R2.

If we focus on the rules of Simplification, R1 and R2, we notice that the effects of the application of any of these rules involve two steps in the derivation: in the former a new literal is added, and in the latter a literal is deleted.

If S is a set of literals, let GS be the set of all the ground instances of S. A literal L is said to be *redundant* with respect to a set of literals S if, for all the ground instances $L\sigma$ of L, it happens that $\{E \mid E \in GS \ \& \ E < L\sigma\} \models L\sigma$. We notice that in our derivations only redundant literals are deleted.

Fact. If in a derivation S_{i+1} is equal to $S_i \setminus \{L\}$, then L is redundant with respect to S_i.

Proof. The claim above is well known if S_{i+1} is obtained from S_i applying one of the rules in Figure 2, and it follows immediately in the case we are applying R1 or R2.

So, as usual, we label with S_∞ the set of literals generated during a derivation δ (in symbols, $S_\infty = \bigcup_i S_i$), and with S_ω the set of persistent literals of δ: $S_\omega = \bigcup_i \bigcap_{j>i} S_j$. We adopt the standard definition for a rule π of the calculus being *redundant* with respect to a set of clauses S whenever, for every ground instance of the rule $\pi\sigma$ it happens that $\{E \mid E \in GS \ \& \ E < C_m\sigma\} \models D\sigma$, where $C_m\sigma$ is the maximal clause in the antecedent, and $D\sigma$ is the consequent of the rule. According to this definition, a derivation w.r.t. $\mathcal{SP}_I$ is *fair* if, for every literal $L_1, L_2, \ldots, L_m \in S_\omega$, every rule that has $L_1, \ldots, L_m$ as premises is redundant w.r.t. S_∞.

Suppose now to take into account a fair derivation δ. We notice that, if a literal L is added at a certain step of the derivation, say S_{i+1}, then L is either a logical consequence of some literals in S_i, or it is a consequence of some literals in S_i and the axioms of the theory T_I. Thus:

Proposition 1. *If the set of persistent literals S_ω contains $\perp$, then S_ω is unsatisfiable in any model of T_I.*

On the other hand, since the reduction rules we can apply during the derivation satisfy the general requirements about the redundancy, we have that:

Proposition 2. *If the set of persistent literals S_ω does not contain $\perp$, then S_ω is satisfiable.*

What remains to show is that this calculus is *refutationally complete* with respect to the models of T_I (namely the structures in which the function s is injective, acyclic and such that 0 does not belong to the image of s). We want to identify in the following at least one case in which the calculus in Figures 1, 2 and 3 is not only refutationally complete w.r.t. T_I, but it is complete, too.

Remark 1. Since the satisfiability of S_ω is equivalent to the satisfiability of S_∞, and since the satisfiability of each step S_{i+1} in the derivation implies the satisfiability of S_i, we have in particular that if S_ω is satisfiable, then S_0 is satisfiable. Moreover, it is immediate to check that the unsatisfiability in the models of T_I of S_ω implies the unsatisfiability of S_0 in the same class of structures. So, in case it happens that the calculus described in Figures 1, 2 and 3 is complete, we can proceed as usual when considering procedures based on saturation methods: an initial set of literals S_0 will be satisfiable (in a model of T_I) if and only if its saturation S_ω does not contain $\perp$.

4.1   Completeness

From now on, we assume that the ordering we consider when performing any application of $\mathcal{SP}_I$ is T_I-*good*:

Definition 6. *We say that an ordering $\succ$ over terms on a signature containing Σ_I is T_I-good whenever it satisfies the following requirements:*

  (i) $\succ$ is a simplification ordering that is total on ground terms;
  (ii) 0 is minimal;
  (iii) whenever two terms t_1 and t_2 are not s-rooted it happens that $\mathsf{s}^{n_1}(t_1) \succ \mathsf{s}^{n_2}(t_2)$ iff either $t_1 \succ t_2$ or ($t_1 \equiv t_2$ and n_1 is bigger than n_2).

Proposition 3. *Assuming T_I-good ordering $\succ$ over terms, if the set of persistent literals S_ω satisfies the following assumptions:*

  – *S_ω does not contain $\bot$,*
  – *s-rooted terms can be maximal just in ground equations in S_ω*

then S_ω is satisfiable in a model of T_I.

Collecting all the results obtained so far, we can conclude that:

Theorem 2. *Let T be a Σ-theory presented as a finite set of unit clauses such that $\Sigma \supseteq \Sigma_I$, and assume to put an ordering over terms that is T_I-good. $\mathcal{SP}_I$ induces a decision procedure for the constraint satisfiability problem w.r.t. $T \cup T_I$ if, for any set G of ground literals:*

  – *the saturation of $Ax(T) \cup G$ w.r.t. $\mathcal{SP}_I$ is finite,*
  – *the saturation of $Ax(T) \cup G$ w.r.t. $\mathcal{SP}_I$ does not contain non-ground equalities whose maximal term is s-rooted.*

4.2   Termination

Proposition 4. *For any set G of ground literals over a signature extending Σ_I, any saturation of G w.r.t. $\mathcal{SP}_I$ is finite.*

Proof. Each step either adds a literal that is smaller than (at least) one literal already present in the saturation, or delete one literal, hence the multiset of literals decreases according to the well-founded ordering $((\succ)^{mul})^{mul}$.

Corollary 1. *$\mathcal{SP}_I$ induces a decision procedure for the constraint satisfiability problem w.r.t. the union of T_I and the theory of equality.*

5   Examples of Integer Offsets Extensions

We investigate theories sharing symbols of T_I in a specific way, thanks to axioms of the form $g(f(\ldots, x, \ldots)) = \mathsf{s}(g(x))$ where f, g are function symbols not occurring in Σ_I. Despite this restricted form of axioms, we are already able to consider interesting examples of Integer Offsets extensions.

5.1   Lists with Length

Let us consider T_{LLI}, the theory of lists endowed with length. T_{LLI} can be axiomatized as the union of the theories T_L, T_ℓ and T_I, where T_I is the theory of Integer Offsets of Example 1 and:[2]

$\boxed{T_L}$ has the multi-sorted signature of the theory of lists: Σ_L is the set of function symbols {nil : LISTS, car : LISTS $\rightarrow$ ELEM, cdr : LISTS $\rightarrow$ LISTS, cons : ELEM $\times$ LISTS $\rightarrow$ LISTS} plus the predicate symbol atom : LISTS, and it is axiomatized as follows:

$$\neg\mathsf{atom}(x) \Rightarrow \mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(x)) = x$$
$$\mathsf{car}(\mathsf{cons}(x, y)) = x \qquad \neg\mathsf{atom}(\mathsf{cons}(x, y))$$
$$\mathsf{cdr}(\mathsf{cons}(x, y)) = y \qquad \mathsf{atom}(\mathsf{nil})$$

$\boxed{T_\ell}$ is the theory that gives the axioms for the function length ℓ : LISTS $\rightarrow$ INT:

$$\ell(\mathsf{nil}) = 0$$
$$\ell(\mathsf{cons}(x, y)) = \mathsf{s}(\ell(y))$$

We want to show that the constraint satisfiability problem for T_{LLI} is decidable via the calculus described in the previous section.

First: Reduction. We start addressing the problem of checking the satisfiability of a constraint w.r.t. T_{LLI}. Let G be a set of ground literals over $\Sigma_{T_{LLI}}$; we can associate to G the set of formulae G' obtained by replacing all the literals in $G \cup \{\mathsf{atom}(\mathsf{nil})\}$ in the form $\neg\mathsf{atom}(t)$ and $\mathsf{atom}(t')$ with respectively $t = \mathsf{cons}(sk_1, sk_2)$ and $\forall x_0, x_1\, t' \neq \mathsf{cons}(x_0, x_1)$, where t and t' are ground terms of sort LISTS and sk_1, sk_2 are fresh constants of the appropriate sort (this is the same reduction used in [2]).

Let now $T_{L'}$ be the subtheory of T_L whose axioms are just the first two (equational) axioms of T_L. We have that:

Proposition 5. *G is satisfiable w.r.t. T_{LLI} if and only if G' is satisfiable w.r.t. $T_{L'} \cup T_\ell \cup T_I$.*

Second: Saturation. According to Proposition 5 and applying at most some standard steps of flattening, we can focus our attention to sets of literals of the following kinds (x is a variable of sort ELEM, y is a variable of sort LISTS, $h, l, a, f, g, l_1, l_2, e, d, e_1, e_2, i, i_1, i_2$ are constants of the appropriate sorts and the symbol $\bowtie$ is a shortening for both $=$ and $\neq$), and the left-hand side of all the literals is the maximal one.

---

[2] All the axioms should be considered as universally quantified.

i.) equational axioms for lists
  a) $\mathsf{car}(\mathsf{cons}(x, y)) = x$;
  b) $\mathsf{cdr}(\mathsf{cons}(x, y)) = y$;
ii.) reduction for $\neg\mathsf{atom}$
  a) $\mathsf{cons}(x, y) \neq h$;
  b) $\mathsf{cons}(x, y) \neq \mathsf{nil}$;
iii.) axioms for the length
  a) $\ell(\mathsf{nil}) = 0$;
  b) $\ell(\mathsf{cons}(x, y)) = \mathsf{s}(\ell(y))$;
iv.) ground literals over the sort LISTS
  a) $\mathsf{cons}(e, l) = c$;

b) $\mathsf{cdr}(f) = g$;
c) $l_1 \bowtie l_2$;
v.) ground literals over the sort ELEM
  a) $\mathsf{car}(h) = d$;
  b) $e_1 \bowtie e_2$;
vi.) ground literals over the sort INT
  a) $\ell(a) = \mathsf{s}^m(i)$;
  b) $\mathsf{s}^m(i_1) \neq \mathsf{s}^n(i_2)$;
  c) $\mathsf{s}^n(i_1) = i_2$;
  d) $i_1 = \mathsf{s}^n(i_2)$.

Let us choose, as ordering over the terms, a LPO ordering $\succ$ whose underlying precedence over the symbols of the signature respects the following requirements:

- $\mathsf{cons} > \mathsf{cdr} > \mathsf{car} > c > e > \ell$ for every constant c of sort LISTS and every constant e of sort ELEM;
- $\ell > i > 0 > \mathsf{s}$ for every constant i of sort INT;

These requirements over the precedence guarantee that every compound term of sort LISTS is bigger than any constant, any compound term over the sort ELEM is bigger than any constant, and that $\succ$ is a T_I-*good* ordering.

We require that the rules in Figures 2 and 3 are applied, whenever possible, before the rules in Figure 1 (in other words we require that the contraction rules have a higher priority).

Proposition 6. *For any set G of ground literals, any saturation of $Ax(T_{LLI}) \cup G$ w.r.t. $\mathcal{SP}_I$ is finite.*

The key observations, in order to prove termination, are that the non-ground set of literals is already saturated, every equation obtained by the application of a rule to ground factors is smaller in the ordering w.r.t. the biggest factor in the antecedent of the rule, and every application of a rule of the calculus to a ground and a non-ground literal produces a ground literal that is smaller than the ground factor. In other terms, every literal produced during the saturation phase is ground and it is strictly smaller than the biggest ground literal in the input set. Since the ordering on the literals is the multiset extension of a terminating ordering, it is terminating too.

Moreover, since in the saturation no non-ground equation whose maximal term is s-rooted is generated, we can conclude by Theorem 2 that $\mathcal{SP}_I$ is a decision procedure for the constraint satisfiability problem w.r.t. T_{LLI}.

5.2 Lists over Integer Elements

Let us consider now lists whose elements are integers. The reduction of Section 5.1 works without any changes, so we can check if the calculus developed in Figures 1, 2 and 3 is still a decision procedure for the constraint satisfiability problem of lists with length and integer elements. We can apply at most some

standard steps of flattening and we focus our attention to sets of literals of the kinds i—iv) defined in Section 5.1 plus the new following one which merges the kinds v—vi) of Section 5.1:

v.) ground literals over the sort INT

a) $\mathsf{car}(h) = \mathsf{s}^n(i)$;
b) $\ell(a) = \mathsf{s}^m(i)$;
c) $\mathsf{s}^m(i_1) \neq \mathsf{s}^n(i_2)$;

d) $\mathsf{s}^n(i_1) = i_2$;

e) $i_1 = \mathsf{s}^n(i_2)$.

Let us put over the symbols of the signature an order that respects the same requirements we have asked in Section 5.1. The same remarks about termination and the shape of the saturated set of the previous section apply also to this case, guaranteeing that $\mathcal{SP}_I$ provides a decision procedure.

5.3   Records with Increment

Let us consider records in which all the attribute identifiers are associated to the same sort INT, and suppose we want to be able to increment by a unity every value stored into the record. To formalize this situation, we can choose a signature as follows: let $Id = \{id_1, id_2, \ldots, id_n\}$ a set of attribute identifiers and let us name REC the sort of records; for every attribute identifier $id_1, id_2, \ldots, id_n$ we have a couple of functions $\mathsf{rselect}_i : \text{REC} \to \text{INT}$ and $\mathsf{rstore}_i : \text{REC} \times \text{INT} \to \text{REC}$; moreover, there is also the increment function $\mathsf{incr} : \text{REC} \to \text{REC}$. The axioms of the theory of integer record with increment, T_{IRI}, are the following:

$\boxed{T_{IRI}}$: for every i, j such that $1 \leq i \neq j \leq n$

$$\mathsf{rselect}_i(\mathsf{rstore}_i(x, y)) = y$$
$$\mathsf{rselect}_j(\mathsf{rstore}_i(x, y)) = \mathsf{rselect}_j(x)$$
$$\wedge_{i=1}^n (\mathsf{rselect}_i(x) = \mathsf{rselect}_i(y)) \Rightarrow x = y \qquad \text{(extensionality)}$$
$$\mathsf{rselect}_i(\mathsf{incr}(x)) = \mathsf{s}(\mathsf{rselect}_i(x))$$

In order to check the satisfiability of a set of ground literals w.r.t. T_{IRI}, we notice that every literal of the kind $r_1 \neq r_2$ is equivalent to a clause of the kind $\bigvee_{i=1}^n \mathsf{rselect}_i(r_1) \neq \mathsf{rselect}_i(r_2)$, so can we substitute every disequation between records with the corresponding clause and then check the satisfiability of the resulting set of clauses by case split.

So we can restrict our attention to sets of literals in which no disequation between records appears. In this case, following the same argument used in [1], it is possible to check the satisfiability forgetting the extensionality axioms (the presence of the function incr does not affect the argument). Thus we are reduced to consider the saturation of sets of literals of the following kind:

i.) equational axioms for records
   a) $\mathsf{rselect}_i(\mathsf{rstore}_i(x,y)) = y$;
   b) $\mathsf{rselect}_j(\mathsf{rstore}_i(x,y)) = \mathsf{rselect}_j(x)$;
   c) $\mathsf{rselect}_i(\mathsf{incr}(x)) = \mathsf{s}(\mathsf{rselect}_i(x))$;
ii.) ground literals over the sort REC
   a) $r_1 = r_2$;
   b) $\mathsf{rstore}_i(r_1, \mathsf{s}^n(k)) = r_2$;

c) $\mathsf{incr}(r_1) = r_2$;
iii.) ground literals over the sort INT
   a) $\mathsf{rselect}_i(r) = \mathsf{s}^n(k)$;
   b) $\mathsf{s}^n(k_1) = k_2$;
   c) $k_1 = \mathsf{s}^n(k_2)$;
   d) $\mathsf{s}^n(k_1) \neq \mathsf{s}^m(k_2)$.

where x is a variable of sort REC, y is a variable of sort INT, and r, r_1, r_2, k, k_1, k_2 are constants of appropriate sorts. As usual, let us consider a LPO ordering over terms such that the underlying precedence over the symbols in the signature satisfies the following requirements: for all i, j in $\{1, \ldots, n\}$, $\mathsf{incr} > \mathsf{rstore}_i$, $\mathsf{rstore}_i > \mathsf{rselect}_j$, $\mathsf{rselect}_i > c$ for every constant c and every constant c is such that $c > 0 > \mathsf{s}$.

Proposition 7. *For any set G of ground literals, any saturation of $Ax(T_{IRI}) \cup G$ w.r.t. $\mathcal{SP}_I$ is finite.*

The completeness of the calculus can be shown relying on the observation that no non-ground literals involving the function symbol s are generated, and that the chosen ordering is a T_I-*good* one.

6   Combination of Theories Sharing Integer Offsets

In the previous section we have collected examples of theories extending the theories of the Integers Offsets T_I and whose constraint satisfiability problem is decidable. We have already noticed that T_I admits a model completion $T_I^\star$ and that it is a Noetherian theory; to guarantee that the theories that have been studied can be combined all together it is sufficient to show that they fully satisfy the requirement of being T_I-compatible and effectively Noetherian extension of T_I.

6.1 T_I-Compatibility

Being for a theory $T \supseteq T_I$ a T_I-compatible theory means that every constraint that is satisfiable w.r.t. T is satisfiable also in a model in which the axiom $\forall x(x \neq 0 \Rightarrow \exists y\, x = \mathsf{s}(y))$ holds. To see that actually it is the case for all the theories considered in Section 5, it is sufficient to check that any model of that theories can always be extended, if needed, adding recursively to each element that is different from (the interpretation of) 0 its predecessor and, in case it is needed, modifying accordingly the remaining part of the structure; and to check that this enlargement does not affect the validity both of the constraints that are verified in the structure and of the axioms of the theory. For example, we consider in [15] the case of the theory of lists over integer elements with length. Using similar (or simpler) arguments as the ones for this case, it is possible to verify that all the theories in Section 5 are T_I-compatible.

6.2   Derivation of T_I-Basis

We have considered Horn Σ'-theories $T' = T \cup T_I$ extending T_I with some theories T axiomatized by unit clauses. We have shown under which assumptions the Superposition Calculus $\mathcal{SP}_I$ is complete in order to check T'-satisfiability of sets of ground literals. Let us show that $\mathcal{SP}_I$ allows us to derive T_I-basis. Assume that $G(\underline{a}, \underline{b})$ is a set of ground literals over an expansion of Σ' with the finite sets of fresh constants $\underline{a}, \underline{b}$. Our claim is the following: if S_ω is the saturation of $Ax(T) \cup G(\underline{a}, \underline{b})$ and assuming a T_I-*good* order over the terms in the signature $\Sigma' \cup \{\underline{a}, \underline{b}\}$ such that every term over the subsignature Σ_I^a is smaller than any term that contains a symbol in $(\Sigma' \setminus \Sigma_I) \cup \{\underline{b}\}$, then the subset of OGS_ω over the signature Σ_I^a, denoted by $\Delta(\underline{a})$, is a T_I-basis. Since T' is a Horn theory, it is convex and so we can focus our attention just over equations instead of positive (ground) clauses.

Proposition 8. *If $l = r$ is an equation over Σ_I^a implied by $T' \cup G(\underline{a}, \underline{b})$, then $l = r$ is already implied by $T_I \cup \Delta(\underline{a})$*

Proof. From what we have shown in the previous paragraphs we have that $T' \cup G(\underline{a}, \underline{b}) \cup \{l \neq r\}$ has a model iff a saturation of $Ax(T) \cup G(\underline{a}, \underline{b}) \cup \{l \neq r\}$ does not contain $\bot$. A saturation of $Ax(T) \cup G(\underline{a}, \underline{b}) \cup \{l \neq r\}$ is equal to a saturation of $S_\omega \cup \{l \neq r\}$. Since S_ω does not contain $\bot$, the only alternative way to derive $\bot$ is by reducing $l \neq r$ to $\bot$ using equations from $\Delta(\underline{a})$, since $l \neq r$ is defined on the signature $\mathsf{s} \cup 0 \cup \underline{a}$ and no equation in S_ω containing a symbol different from $\mathsf{s}, 0, \underline{a}$ can be used to rewrite a term on signature $\mathsf{s}, 0, \underline{a}$ by our choice of the reduction ordering. Hence the saturation of $Ax(T) \cup G(\underline{a}, \underline{b}) \cup \{l \neq r\}$ does not contain $\bot$ iff the saturation of $\Delta(\underline{a}) \cup \{l \neq r\}$ does not contain $\bot$. Since $\Delta(\underline{a}) \cup \{l \neq r\}$ is a set of ground literals, Theorem 2 applies, and the saturation of $\Delta(\underline{a}) \cup \{l \neq r\}$ does not contain $\bot$ iff $\Delta(\underline{a}) \cup \{l \neq r\}$ has a T_I-model.

7   Conclusion

We have shown how to apply a superposition calculus to build decision procedures for some theories sharing Integer Offsets. These theories and the related decision procedures satisfy all the requirements for their applications in a non-disjoint combination framework. To the best of our knowledge, this paper is the first contribution showing the interest of a superposition calculus for non-disjoint combinations. This paper paves the way of using non-disjoint combinations (with a shared fragment of Arithmetics) in the context of verification. There are several research directions we want to investigate. Currently, the soundness of the superposition calculus is proved manually for each theory considered in the paper. It would be very interesting to have an automatic proof mechanism using for instance a meta-saturation calculus [13]. Moreover, the considered fragment of Arithmetics is not very expressive and we have some limitations on the form of axioms we are able to handle. Further works are needed to go beyond these restrictions.

References

1. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Transactions on Computational Logic 10(1)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)
3. Bonacina, M.P., Echenim, M.: On variable-inactivity and polynomial T-satisfiability procedures. Journal of Logic and Computation 18(1), 77–96 (2008)
4. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., Ranise, S., van Rossum, P., Sebastiani, R.: Efficient theory combination via boolean search. Information and Computation 204(10), 1493–1525 (2006)
5. Bradley, A., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007)
6. 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)
7. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (1972)
8. Ghilardi, S.: Model theoretic methods in combined constraint satisfiability. Journal of Automated Reasoning 33(3-4), 221–249 (2004)
9. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Noetherianity and combination problems. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 206–220. Springer, Heidelberg (2007); extended version at `http://homes.dsi.unimi.it/~zucchell/publications/conference/GhiNiRaZu-FroCoS-07.pdf`
10. Ghilardi, S., Nicolini, E., Zucchelli, D.: A comprehensive combination framework. ACM Transactions on Computational Logic 9(2), 1–54 (2008)
11. Kirchner, H., Ranise, S., Ringeissen, C., Tran, D.-K.: On superposition-based satisfiability procedures and their combination. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 594–608. Springer, Heidelberg (2005)
12. Krstić, S., Goel, A., Grundy, J., Tinelli, C.: Combined satisfiability modulo parametric theories. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 618–631. Springer, Heidelberg (2007)
13. Lynch, C., Tran, D.-K.: Automatic Decidability and Combinability Revisited. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 328–344. Springer, Heidelberg (2007)
14. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transaction on Programming Languages and Systems 1(2), 245–257 (1979)
15. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Satisfiability Procedures for Combination of Theories Sharing Integer Offsets. Report, INRIA, RR-6697 (2008)
16. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. I, pp. 371–443. Elsevier Science, Amsterdam (2001)
17. Shostak, R.E.: Deciding combinations of theories. J. of the ACM 31, 1–12 (1984)
18. Zucchelli, D.: Combination Methods for Software Verification. PhD thesis, Università degli Studi di Milano and Université Henri Poincaré - Nancy 1 (2008)

Bridging the Gap Between Model-Based Development and Model Checking⋆

Steven P. Miller

Rockwell Collins, Cedar Rapids IA 52498, USA
spmiller@rockwellcollins.com

Abstract. The growing power of model checking is making it feasible
to use formal verification for important classes of software systems. How-
ever, for this to be practical it is necessary to bridge the gap between the
commercial modeling tools industrial developers prefer to use and the
input languages of the formal verification tools. This paper describes a
translator framework that makes it possible to use several popular for-
mal verification tools with commercial modeling tools. The practicality of
this approach is illustrated by four case studies in which model checking
was successfully used in the development of avionics software.

1 Introduction

Great strides have been made in the development of model checking tools over the
last few years. However, there have been relatively few instances reported of their
successful application to industrial problems outside of the realm of hardware
engineering. In fact, software and system engineers are often completely unaware
of the opportunities these tools offer.

One of the main reasons for this has been the difficulty of producing software
or system design models that can be analyzed by these tools. Typically, users
of a model checker must first create a separate model in the input language
of the model checker that they believe replicates the behavior of the original
design. Besides introducing significant cost and delay, this also undermines the
developer's confidence in the analysis since it is not performed on the actual
code or design.

This situation is rapidly changing with the growing popularity of Model-Based
Development (MBD) for the design of embedded systems. Tools such as MAT-
LAB Simulink® [1] and Esterel Technologies SCADE Suite™ [2] are achieving
widespread use in the avionics and automotive industry. The graphical models
produced by these tools provide a formal, or nearly formal, specification that is
often amenable to formal analysis.

This paper describes a translator framework developed by Rockwell Collins and the Critical Systems Research Group at the University of Minnesota that bridges this gap and allows production Simulink and SCADE models to be automatically translated to a variety of popular model checkers and theorem provers. Four case studies are presented in which model checking was used to find errors in early requirements and design models, sometimes years before the final code could be integrated and tested on a system rig.

2   Background

The value proposition for formal verification is changing due to the convergence of two trends, the growing popularity of Model-Based Development for the development of embedded systems and the growing power of model checkers. This section provides a brief introduction to Model-Based Development and to model checking.

2.1   Model-Based Development

Model-Based Development refers to the use of domain specific, graphical modeling languages that can be executed and analyzed before the actual system is built. MBD allows developers to create a model of a system, execute it on their desktop, analyze it with automated tools for the required behavior, and use it to automatically generate code and test cases. In the automotive and avionics industry, MBD generally refers to the use of synchronous data flow languages such as MATLAB Simulink or Esterel Technologies SCADE Suite. Synchronous languages latch their inputs at the start of a computation step, compute their outputs and the next system state as a single atomic step, and communicate between components using data flow signals. This differs from the more general class of modeling notations that include support for asynchronous execution of components and communication using message passing.

2.2   Model Checking

Model checkers are formal verification tools that evaluate an input model to determine if it satisfies a given set of properties [3]. A model checker will consider every possible combination of inputs and state, making the verification equivalent to exhaustive testing of the model. If a property is not true, the model checker will produce a counterexample showing how the property can be falsified. While model checkers cannot be used to verify as large a class of models as theorem provers, they are often easier to use and in many cases can provide results in seconds or minutes. This makes them very attractive for use in industrial settings where software designers may not have the expertise or the time to complete a proof using a mechanical theorem prover.

There are many types of model checkers, each with their own strengths and weaknesses. Explicit state model checkers such as SPIN [4] construct a searchable representation of the design model and store a representation of each state visited. Implicit state (symbolic) model checkers such NuSMV [5] use compact

representations (such as Binary Decision Diagrams) of sets of states to describe regions of the model state space that satisfy the properties being evaluated. This often allows them to handle much larger state spaces than explicit state model checkers. Satisfiability modulo theories (SMT) model checkers such as SAL [6] and Prover® Plug-In [7] use a form of induction to reason about models containing real numbers and unbounded arrays. While SMT-based model checkers can deal with infinite state systems, their properties need to written in such a way that they can be proven by induction over an unfolding of the state transition relationship. For this reason, they tend to be more difficult to use than explicit and implicit state model checkers.

3 The Translator Framework

To bridge the gap between industrial modeling tools and some of the more popular model checkers and theorem provers, Rockwell Collins and the Critical Systems Research Group at the University of Minnesota developed a product family of translators [8] as part of the NASA Aviation Safety Program. An overview of this translator framework is shown in figure 1.
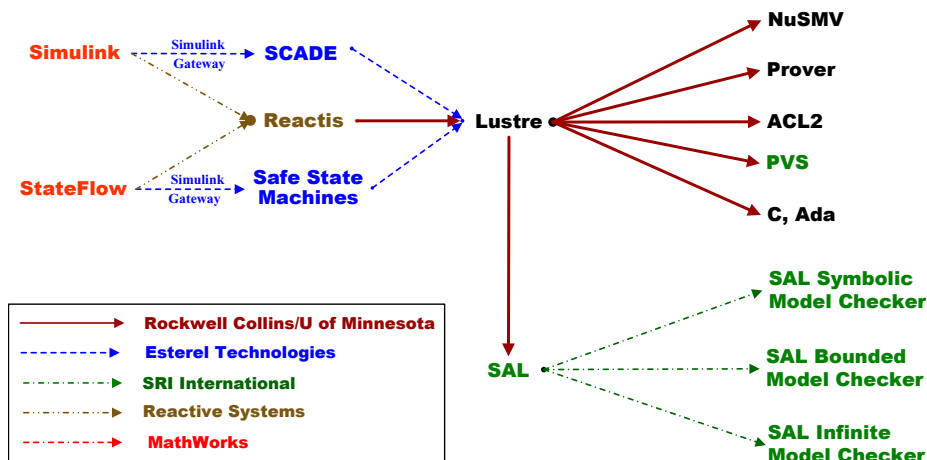
Fig. 1. Translator Framework

The translators work primarily with the Lustre formal specification language [9], but this is hidden from the tool users. The typical user first creates a model in Simulink, StateFlow, or SCADE Suite. Since Lustre is the underlying specification language of SCADE, the initial translation into Lustre is immediate for SCADE models. Simulink and StateFlow users can translate their models into Lustre using either the Simulink Gateway provided by Esterel Technologies or by importing their models into the Reactis® [10] test case generator developed by Reactive Systems and using a translator developed by Rockwell Collins. While Simulink does not have a full, formal semantics, developers of safety-critical

systems routinely restrict themselves to a safe subset of the language and it is usually possible to assign a formal semantics to this subset.

Once in Lustre, the specification is read into an abstract syntax tree (AST) and a number of transformation passes are applied to it. Each transformation pass produces a valid Lustre AST that is syntactically closer to the target specification language and preserves the semantics of the original Lustre specification. This allows all Lustre type checking and analysis tools to be used after each transformation pass. When the AST is sufficiently close to the target language, a pretty printer is used to output the target specification.

The translator framework is actually a product family of translators in that many transformation passes are reused in the translators for each target language. Pre-conditions for each transformation specify the properties a Lustre specification must satisfy for the translation to be valid and post-conditions define the properties of the generated specification. Reuse of the transformation passes makes it much easier to support a variety of target languages and allows new translators to be developed in a matter of days. The number of transformation passes depends on how similar the source and target languages are and on how many optimizations need to be made. Currently, the number of transformation passes ranges from a dozen for a simple C code generator to over sixty for an optimized translation to NuSMV.

The translators produce highly optimized models appropriate for the target language. For example, when translating to the NuSMV model checker, the translator produces a specification that is difficult for a human to read, but very efficient for model checking. When translating to the PVS theorem prover, the specification is optimized for readability and to support the development of proofs in PVS. When generating executable C code, the translation is optimized for execution speed on the target processor. Many of these optimizations can have a dramatic effect on the target analysis tools. For example, optimization passes incorporated into the NuSMV translator reduce the time required for NuSMV to check one model from over 29 hours to less than a second, an improvement of over five orders of magnitude.

Tools have also been developed to present the counter examples produced by the model checkers into two formats that are easier to understand. The first is a simple spreadsheet format that shows the inputs and outputs of the model for each step of the counter example. The second is a test script that can be used to step the Reactis tool forward and backward through the counter example. As shown in figure 1, the translator framework currently supports input models written in MATLAB Simulink and Stateflow and Esterel Technologies SCADE Suite and generates models for the NuSMV, SAL, and PROVER model-checkers, the PVS and ACL2 theorem provers, and C and Ada source code.

4   Case Studies

This section describes four case studies in which model checking was used to find errors in early requirements and design models. The use of industrial models has proven invaluable in selecting the features to be added to the translator framework.

4.1 FCS 5000 Mode Logic

The first application of model checking to an actual product at Rockwell Collins was to the mode logic of the FCS 5000 Flight Control System [11]. The FCS 5000 is a family of Flight Control Systems (FCS) developed by Rockwell Collins for use in business and regional jet aircraft. The Flight Guidance System (FGS) is a component of the FCS that compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The mode logic is a component of the FGS that determines which lateral and vertical flight modes are armed and active at any time.

While inherently complex and difficult to get right, the mode logic consists almost entirely of Boolean and enumerated types. As described in [11], Rockwell Collins developed an in-house format that produces a very compact specification of the mode logic that can be directly implemented in Simulink. This made the FCS 5000 mode logic ideally suited for analysis using the translator framework and a symbolic model checker such as NuSMV.

The mode logic analyzed consisted of five mode transitions diagrams with a total of 36 modes, 172 events, and 488 transitions. Changes in the state of each mode diagram affect at least one, and often more, of the other mode diagrams. While each individual diagram is straightforward to understand, grasping all the interactions between them can be difficult. In fact, the most interesting requirements to be checked defined relationships to be maintained between the mode machines, for example, ensuring that the active vertical flight mode was not "Approach" unless the active lateral flight mode was already "Approach".

Analysis of a very early specification of the FCS 5000 mode logic with NuSMV found 26 errors in the mode logic. Seventeen of these were found by the model checker, six were found in the process of translating the informal requirements into the Simulink model, and three were found during inspections performed to develop the properties to be checked. Of the 17 errors found using the model checker, 13 were classified as being possible to be missed by traditional verification techniques such as testing and inspections, and one was classified as being likely to be missed by traditional techniques.

One of the main advantages of this analysis was that it could be done early in the development process when the requirements were still under development. Finding and correcting errors at this stage is far more cost effective than waiting until executable code is ready for unit and integration testing.

4.2 ADGS-2100 Window Manager

One of the largest and most successful applications of model checking at Rockwell Collins was to the ADGS-2100 Adaptive Display and Guidance System Window Manager [12]. In modern aircraft, the primary way that aircraft status is provided to the pilots is through computerized display panels in the cockpit. These panels replace the dozens of mechanical switches and dials found in earlier aircraft and present a unified interface to critical flight information.

The ADGS-2100 is a Rockwell Collins product that provides the heads-down and heads-up displays and display management software for next generation commercial aircraft. The pilots can switch each panel between several different displays of information such as primary flight displays, navigational maps, aircraft system status, and flight checklists. However, some information is considered critically important and must always be displayed. For this reason, the ADGS-2100 provides redundant implementations of all its critical functions.

The Window Manager (WM) ensures that data from the different displays applications is routed to the correct display panel. In normal operation, the WM determines which applications are being displayed in response to the pilot selections. However, in the case of a component failure, the WM decides which information is most critical and routes this information from one of the redundant sources to the most appropriate display panel. The WM is essential to the safe flight of the aircraft. If the WM contains logic errors, critical flight information could be made unavailable to the flight crew.

Like the FCS 5000 mode logic, the WM is specified in Simulink using only Booleans and enumerated types, but it is surprisingly complex. The WM is composed of five main components that can be analyzed independently. As shown in table 1, these five components contain a total of 16,117 primitive Simulink blocks that are grouped into 4,295 instances of Simulink subsystems. The reachable state space of the five components ranges from 9.8×10^9 to 1.5×10^{37} states.

Table 1. Window Manager Analysis Results

Component	Subsystem Instances	Basic Blocks	Reachable State Space	Number of Properties	Errors Found
GG	2,831	10,699	9.8×10^9	43	56
PS	144	398	4.6×10^{23}	152	10
CM	139	1,009	1.2×10^{17}	169	10
DUF	879	2,941	1.5×10^{37}	115	8
MFD	301	1,100	6.8×10^{31}	84	14
Totals	4,295	16,117		563	98

To begin the analysis, a set of properties that formally state the WM requirements were develped in CTL, one of the property specification languages of NuSMV. Developing the properties to be checked was a gradual process of studying the WM requirements and talking with the WM developers. Some properties were straightforward to write. For example, in a simplified version of the WM with only two Display Units (DU), the requirement

> *If a DU is available, then it shall display some application.*

would be stated as the two CTL properties

```
AG(LEFT_DU_AVAILABLE -> LEFT_DU_APPLICATION != BLANK)

AG(RIGHT_DU_AVAILABLE -> RIGHT_DU_APPLICATION != BLANK)
```

Other properties required discussion with the WM developers to clarify nuances or resolve the ambiguity inherent in English textual requirements.

At the start of the project, the translator chain did not work with the version of Simulink being used by the development team and the models required several hours of hand tweaking before they could be translated from Simulink to NuSMV. As a result, the early analysis was done entirely by the model checking team.

However, as the project progressed, improvements were made to the tool chain so that the translation only took a few minutes and was completely automated. Also, optimizations to the translator reduced the time required for NuSMV to check each property to roughly 20 seconds. Gradually, the developers began to see that model checking could find errors faster, more easily, and more thoroughly than testing or reviews. This motivated them to start writing and checking CTL properties on their own. Eventually, the developers completely took over the model checking and began relying on the model checking team only for consultation and tool improvements.

Ultimately, 563 properties about the WM were developed and checked, and 98 design errors in the model were found and corrected (see table 1). As with the FGS mode logic, this verification was done early in the design process as the design and the requirements were still evolving. In fact, by the end of the project, the WM developers were checking the properties several times each day, usually after each design change.

4.3 CerTA FCS Phase I

The third case study was sponsored by the Air Force Researh Labs (AFRL) Wright Patterson RD Directorate under the Certification Technologies for Advanced Flight Control Systems (CerTA FCS) program [13]. In this study, the translation framework and model checking tools were applied to the Operational Flight Program (OFP) for an Unmanned Aerial Vehicle (UAV) created by Lockheed Martin Aero. The OFP is an adaptive flight control system that modifies its behavior in response to flight conditions.

Phase I of the project focused on investigating the roles of testing and formal verification, and in particular, determining if formal verification could be used to replace some tesing. To this end, two verification teams were set up. One team, based at Lockheed Martin, focused on traditional testing of the OFP. The other team, based at Rockwell Collins, focused on the use of model checking. Neither team communicated directly with the other team and both teams started with identical models and specifications of the requirements.

To ensure the effectiveness of testing was being compared to a mature formal verification technology, the model checking in Phase I was restricted to the Redundancy Management (RM) logic of the OFP. Like the FCS 5000 mode logic and the ADGS-2100 WM, the RM logic is based almost entirely on Boolean and enumerated types. This makes it ideal for analysis with a BDD-based model checker such as NuSMV. However, the RM logic also contained several model constructs that had not been encountered in the FCS 5000 mode logic and the

ADGS-2100 WM, including Stateflow models and truth tables. Extensions to the translator framework to support these features took about two thirds of the total time spent model checking the RM logic. However, the Lockheed Martin team also made comparable investments in enhancing their testing environment. These one time, non-recurring costs were factored out of the final comparison of the effectiveness of testing and model checking.

Like the ADGS-2100 WM, the RM logic is organized into three components that could be analyzed individually (see table 2). While these components are smaller than those in the ADGS-2100 WM, they are replicated once for each of the ten control surfaces on the aircraft and collectively represent a significant portion of the OFP logic.

Table 2. OFP Redundancy Manager Analysis Results

Component	Subsystem Instances	Basic Blocks	Charts/ Transitions/ TT Cells	Reachable State Space	Number of Properties	Errors Found
Triplex Voter	10	96	3/35/198	6.0×10^{13}	43	5
Failure Processing	7	42	0/0/0	2.1×10^4	6	3
Reset Manager	6	31	2/26/0	1.3×10^{11}	8	4
Totals	23	169	5/61/198		62	12

To compare the effectiveness of model checking and testing at discovering errors, the formal verification team developed a total of 62 properties from the OFP requirements. While these properties only partially specifed the required behavior of the RM logic, checking them with the model checker uncovered 12 errors in the RM logic. Of these 12 errors, four were classified as severity 3 (only severity 1 and 2 can affect the safety of flight), two were classified as severity 4, two resulted in requirements changes, one was redundant, and three resulted from requirements that had not yet been implemented in that release of the software.

In similar fashion, the testing team developed a series of tests from the same OFP requirements. Even though the testing team invested almost half again as much time in testing as the formal verification team spent in model checking, testing failed to find any errors, including those found through model checking. The conclusion of both teams was that in this case, model checking was more effective than testing in finding design errors.

4.4   CerTA FCS Phase II

The purpose of Phase II of the CerTA FCS project was to investigate whether model checking could be used to verify large, numerically intensive models. In this study, the translation framework and model checking tools were used to verify important properties of the Effector Blender (EB) logic of an OFP for a UAV similar to that verified in Phase I. The EB is a central component of the OFP that generates the actuator commands for the aircraft's six control

surfaces. It is a large, complex piece of logic that repeatedly manipulates a 3×6 matrix of floating point numbers. It inputs 32 floating point inputs and a 3×6 matrix of floating point numbers and outputs a 1×6 matrix of floating point numbers. It contains over 2,000 basic Simpulink blocks organized into 166 Simulink subsystems, many of which are Stateflow models.

Because of its extensive use of floating point numbers and enormous state space, the EB cannot be verified using a BDD-based model checker such as NuSMV. Instead, the EB was analyzed using the Prover SMT-solver from Prover Technologies. Even with the additional capabilities of Prover, several new issues had to be addressed in Phase II, the hardest being dealing with floating point numbers.

While Prover has powerful decision procedures for linear arithmetic with real numbers and bit-level decision procedures for integers, it does not have decision procedures for floating point numbers. Translating the floating point numbers into real numbers was rejected since much of the arithmetic in the EB is inherently non-linear. Also, the use of real numbers would mask floating point arithmetic errors such as overflow and underflow.

Instead, the translator framework was extended to convert floating point numbers to fixed point numbers using a scaling factor provided by the OFP designers. The fixed point numbers were then converted to integers using bit-shifting to preserve their magnitude. While this allowed the EB to be verifed using Prover's bit-level integer decision procedures, the results were unsound due to the loss of precision. Even so, if errors were found in the verified model, it was very likely that they would also be found in the original model. This allowed the verification to be used as a highly effective debugging step, even though it did not guarantee correctness.

Determining what properties to verify was also a difficult problem. The requirements for the EB are actually specified for the combination of the EB and the aircraft model, but checking both the EB and the aircraft model exceeded the capabilities of the Prover model checker. After extensive consultation with the OFP designers, the verification team decided to verify whether the six actuator commands would always be within a dynamically computed upper and lower limit. Violation of these properties would indicate a design error in the EB logic.

Even with these adjustments, the EB logic was large enough that it had to be decomposed into a hierarchy of components several levels deep. The leaf nodes of this hierarchy were then verified using Prover and their composition was manually verified using through simple manual proofs. This approach also ensured that unsoundness could not be introduced through circular reasoning since Simulink enforces the absence of cyclic dependencies between atomic subsystems.

Ultimately, five errors in the EB design logic were discovered and corrected through verification of these properties. In addition, several potential errors that were being masked by defensive design practices were found and corrected.

5   Conclusions and Future Directions

The case studies described in this paper demonstrate that model checking can be effectively used to find errors early in the development process for many

classes of models. In particular, even very complex models can be verified with BDD-based model checkers if they consist primarily of Boolean and enumerated types. Every industrial system we have studied contains large portions of logic that either meet this constraint or that can be made to meet it with some alteration.

For this class of models, the tools are simple enough for developers to use them routinely and without extensive training. In our experience, a single day of training and a low level of ongoing mentoring is usually sufficient. This also makes it practical to perform model checking early in the development process while a model is still changing. Running a set of properties after each model revision is a quick and easy way to see if anything has been broken. We encourage our developers to "check your models early and check them often." The time spent model checking is recovered several times over by avoiding rework during unit and integration testing.

Since model checking examines every possible combination of input and state, it is also far more effective at finding design errors than testing, which can only check a small fraction of the possible inputs and states. When combined with the ease of use discussed above, this makes it very cost effective approch to defect detection. As demonstrated by the CerTA FCS Phase I case study, it can be more cost effective than testing.

However, there are still many areas for further research. As illustrated in the CerTA FCS Phase II study, numerically intensive models still pose a challenge for model checking. In fact, even a handful of integers can render BDD-based model checking ineffective. SMT-based model checkers hold great promise for verification of these systems, but the need to write properties that can be verified through induction over the state transition relation make them more difficult for developers to use. More work is needed to make them simpler and more intuitive.

Most industrial models used to generate code make extensive use of floating point numbers. As discussed in the CerTA FCS Phase II study, simply using real numbers instead of floating point numbers may not be acceptable, either because of the inherent non-linearity of the system or because of the masking of floating point arithmetic errors. Other models, particularly those that deal with spacial relationships such as navigation, make extensive use of trigonometric and other transcendental functions. A simple way of model checking such systems would be very helpful.

It can also be difficult to determine how many properties need to be checked. Our experience has been that checking even a few properties will find errors, but that checking more properties will find more errors. Unlike testing for which many objective coverage criteria have been developed [14], completeness criteria for properties do not seem to exist. Techniques for developing or measuring the adequacy of a set of properties would be very helpful, particularly when seeking certification credit for the use of formal methods.

As discussed in the CerTA FCS Phase II case study, the verification of very large models may be achieved by using model checking on subsystems and more traditional reasoning to compose the subsystems. Combining model checking and

theorem proving in this way could be a very effective approach, but introducing even this limited use of theorem proving into an industrial development process poses many challenges unless it can be made quicker and more intuitive.

Finally, most safety critical systems must be designed using redundancy to meet their reliability requirements. These systems are typically implemented as globally asynchronous/locally synchronous systems in which synchronous components, each with their own clock, communicate asynchronously with each other. Verification of such quasi-synchronous systems [15] pose many challenges to model checking. However, these are also precisely the type of systems that would benefit the most from a formal approach to verification.

References

1. The Mathworks, Simulink Product Description, `http://www.mathworks.com`
2. Esterel Technologies, SCADE Suite Product Description,
   `http://www.estereltechnolgies.com`
3. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (2001)
4. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
5. The NuSMV Model Checker, `http://nusmv.irst.itc.it`
6. SRI International, Symbolic Analysis Laboratory, `http://sal.csl.sri.com`
7. Prover Technology, Prover Plug-In Product Description, `http://www.prover.com`
8. Miller, S., Tribble, A., Whalen, M., Heimdahl, M.: Proving the Shalls. International Journal on Software Tools for Technology Transfer (STTT) 8(4-5), 303–319 (2006)
9. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Dataflow Programming Language Lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
10. Model-Based Testing and Validation with Reactis,
    `http://www.reactive-systems.com`
11. Miller, S., Anderson, E., Wagner, L., Whalen, M., Heimdahl, M.: Formal Verification of Flight Critical Software. In: Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit, AIAA-2005-6431. American Institute of Aeronautics and Astronautics (2005)
12. Whalen, M., Innis, J., Miller, S., Wagner, L.: ADGS-2100 Adaptive Display & Guidance System Window Manager Analysis. NASA Contractor Report CR-2006-213952 (2006), `http://shemesh.larc.nasa.gov/fm/fm-collins-pubs.html`
13. Whalen, M., Cofer, D., Miller, S., Krogh, B., Storm, W.: Integration of Formal Analysis into a Model-Based Software Development Process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)
14. Chilenski, J., Miller, S.: Applicability of Modified Condition/Decision Coverage to Software Testing. IEE Software Engineering Journal 9(5), 193–200 (1994)
15. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Di Natale, M.: Implementing Synchronous Models on Loosely Time Triggered Architectures. IEEE Transactions on Computers 57(10), 1300–1314 (2008)

Author Index