Neil D. Jones
Markus Müller-Olm (Eds.)

LNCS 5403

Verification, Model Checking, and Abstract Interpretation

10th International Conference, VMCAI 2009
Savannah, GA, USA, January 2009
Proceedings

Lecture Notes in Computer Science 5403

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

Neil D. Jones Markus Müller-Olm (Eds.)

Verification, Model Checking, and Abstract Interpretation

10th International Conference, VMCAI 2009
Savannah, GA, USA, January 18-20, 2009
Proceedings

Springer

Volume Editors

Neil D. Jones
(Emeritus University of Copenhagen)
2960 Rungsted, Denmark
E-mail: neil@diku.dk

Markus Müller-Olm
Westfälische Wilhelms-Universität
Institut für Informatik
48149 Münster, Germany
E-mail: markus.mueller-olm@uni-muenster.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): F.3.1-2, D.3.1, D.2.4

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

ISSN 0302-9743
ISBN-10 3-540-93899-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-93899-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12598373 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009), held in Savannah, Georgia, USA, January 18–20, 2009.

VMCAI 2009 was the 10th in a series of meetings. Previous meetings were held in Port Jefferson 1997, Pisa 1998, Venice 2002, New York 2003, Venice 2004, Paris 2005, Charleston 2006, Nice 2007, and San Francisco 2008.

VMCAI centers on state-of-the-art research relevant to analysis of programs and systems and drawn from three research communities: verification, model checking, and abstract interpretation. A goal is to facilitate interaction, cross-fertilization, and the advance of hybrid methods that combine two or all three areas. Topics covered by VMCAI include program verification, program certification, model checking, debugging techniques, abstract interpretation, abstract domains, static analysis, type systems, deductive methods, and optimization.

The Program Committee selected 24 papers out of 72 submissions based on anonymous reviews and discussions in an electronic Program Committee meeting. The principal selection criteria were relevance and quality.

VMCAI has a tradition of inviting distinguished speakers to give talks and tutorials. This time the program included three invited talks by:

- E. Allen Emerson (University of Texas at Austin) on "Model Checking: Progress and Problems"
- Aarti Gupta (NEC Labs, Princeton) on "Model Checking Concurrent Programs"
- Mooly Sagiv (Tel-Aviv University) on "Thread Modular Shape Analysis"

There were also two invited tutorials by:

- Byron Cook (Microsoft Research, Cambridge) on "Proving Program Termination and Liveness"
- Véronique Cortier (LORIA, CNRS, Nancy) on "Verification of Security Protocols".

We would like to thank the members of the Program Committee and the subreviewers for their dedicated effort in the paper selection process that was crucial for the quality of the conference. Our thanks go also to the Steering Committee members for helpful advice, in particular to Dave Schmidt and Lenore Zuck for their invaluable efforts in the conference organization. VMCAI 2009 was colocated with POPL 2009 and we thank Yitzhak Mandelbaum, who served as our interface to the POPL organizers, for the good cooperation. We are also grateful to Andrei Voronkov, whose EasyChair system was tremendously helpful for the submission and paper selection process and compilation of the proceedings. We thank Peter Lammich for his help in setting up the website.

VMCAI was held in cooperation with ACM (Association for Computing Machinery), who we wish to thank for help with local arrangements. VMCAI 2009 was sponsored by EAPLS (European Association for Programming Languages and Systems) and Microsoft Research.

January 2009

Neil Jones
Markus Müller-Olm

Organization

Program Chairs

Neil Jones	University of Copenhagen, Denmark
Markus Müller-Olm	Universität Münster, Germany

Program Committee

Christel Baier	Technische Universität Dresden, Germany
Ahmed Bouajjani	Université Paris 7, France
Michael Codish	Ben-Gurion University of the Negev, Israel
Patrick Cousot	École Normale Supérieure, Paris, France
Javier Esparza	Technische Universität München, Germany
Klaus Havelund	NASA JPL, USA
Michael Huth	Imperial College London, UK
Bengt Jonsson	Uppsala Universitet, Sweden
Sorin Lerner	University of California, San Diego, USA
Francesco Logozzo	Microsoft Research, Redmond, USA
Pete Manolios	Northeastern University, Boston, USA
Amir Pnueli	New York University, USA
C. R. Ramakrishnan	SUNY at Stony Brook, NY, USA
Andrey Rybalchenko	MPI for Software Systems, Germany
Helmut Seidl	Technische Universität München, Germany
Henny Sipma	Stanford University, USA
Carolyn Talcott	SRI International, Menlo Park, CA, USA
Greta Yorsh	IBM T.J. Watson Research Center, NY, USA

Steering Committee

Agostino Cortesi	Università Ca' Foscari di Venezia, Italy
Patrick Cousot	École Normale Supérieure, France
E. Allen Emerson	University of Texas at Austin, USA
Giorgio Levi	University of Pisa, Italy
Andreas Podelski	Universität Freiburg, Germany
Thomas W. Reps	University of Wisconsin at Madison, USA
David Schmidt	Kansas State University, USA
Lenore Zuck	University of Illinois at Chicago, USA

External Reviewers

Elvira Albert
Eyad Alkassar
Puri Arenas
Eugene Asarin
Gogul Balakrishnan
Samik Basu
Amir Ben-Amram
Josh Berdine
Julien Bertrane
Dirk Beyer
Olivier Bouissou
Tomáš Brázdil
Benjamin Chambers
Jed Davis
Giorgio Delzanno
Peter Dillinger
Jérôme Feret
Andrea Flexeder
John Gallagher
Pierre Ganty
Thomas Martin Gawlitza
Samir Genaim
Roberto Giacobazzi
Alex Groce
Marcus Größer
Peter Habermehl
Tingting Han
Christine Hang
Bertrand Jeannet
Stefan Kiefer
Andy King
Jörg Kreiker
Pascal Lafourcade
François Laroussinie
Martin Leucker
Tal Lev-Ami
Michael Luttenberger
Alexander Malkis
Roman Manevich
Matthew Might
Antoine Miné
Rosemary Monahan
Juan Antonio Navarro Pérez
Joachim Parrow
Doron Peled
Michael Petter
David Pichardie
Ruzica Piskac
Nir Piterman
Corneliu Popeea
Polyvios Pratikakis
Francesco Ranzato
Tamara Rezk
Noam Rinetzky
Xavier Rival
Stefan Schwoon
Koushik Sen
Sharon Shoham
A. Prasad Sistla
Jan-Georg Smaus
Harald Søndergaard
Fausto Spoto
Scott Stoller
Dejvuth Suwimonteerabuth
Stavros Tripakis
Yih-Kuen Tsay
Aaron Turon
Margus Veanes
Arnaud Venet
Vesal Vojdani
Tomáš Vojnar
Björn Wachter
Andrzej Wasowski
Eran Yahav
Wang Yi
Suan Yong
Anna Zaks
Damiano Zanardini

Table of Contents

Invited Talks

Invited Tutorials

Submitted Papers

Model Checking: Progress and Problems

E. Allen Emerson

University of Texas at Austin

Abstract. Model checking is an automatic method of verifying finite state concurrent programs. The use of temporal logic and related frameworks to specify correctness has greatly facilitated simply thinking about the verification problem. Despite early worries about the intractability of state explosion, nowadays it can often be ameliorated, permitting verification of enormously large systems in practice. Important techniques include abstraction and compact (symbolic) representation.

Unfortunately, none of these techniques scale well beyond a certain range. Nor is temporal logic universally viewed as a natural specification framework. We will discuss some possible ways to enhance efficiency and expressiveness of model checking.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, p. 1, 2009.

Model Checking Concurrent Programs

Aarti Gupta

NEC Laboratories America, Princeton
agupta@nec-labs.com

Abstract. With the growth of multi-core processing and concurrent programming in modern computing systems, there is a great need to develop effective verification techniques for concurrent programs. Static analysis techniques have been shown effective for finding data races, but suffer from a general problem of too many false alarms. Dynamic techniques like testing have also shown promise, but provide limited coverage over the state space including all possible thread interleavings. Model checking alone cannot scale. However, it works better in combination with these techniques, with the potential of finding real error traces on one hand and better coverage on the other. In this talk, I will describe our recent advances in concurrent dataflow analysis, symbolic model checking with partial order reduction, and dynamic techniques for verifying concurrent programs. These techniques have been implemented in a unified verification platform, currently targeted at multi-threaded C programs. I will also report on our experiences on some challenging examples from the public domain and the industry.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, p. 2, 2009.

Thread-Modular Shape Analysis

Mooly Sagiv

Tel-Aviv University

Abstract. Thread-modular static analysis of concurrent systems abstracts away the correlations between the local variables (and program locations) of different threads. This idea reduces the exponential complexity due to thread interleaving and allows us to handle programs with an unbounded number of threads.

Thread-modular static analyses face a major problem in simultaneously requiring a separation of the reasoning done for each thread, for efficiency purposes, and capturing relevant interactions between threads, which is often crucial to verify properties. Programs that manipulate the heap complicate thread-modular analysis. Naively treating the heap as part of the global state, accessible by all threads, has several disadvantages since it still admits exponential blow-ups in the heap and is not precise enough to capture things like ownership transfers of heap objects. An effective thread-modular analysis needs to determine which parts of the heap are owned by which threads to obtain a suitable thread-modular state abstraction.

I will present new thread-modular analysis techniques and adaptations of thread-modular analysis for programs which manipulate the heap. It is shown that the precision of thread-modular analysis is improved by tracking some correlations between the local variables of different threads. I will also describe techniques for reducing the analysis time for common situations. A key observation for handling the heap is using notions of separation and more generally subheaps in order to abstract away correlations between the properties of subheaps.

This is a joint work with Josh Berdine, Byron Cook, Alexey Gotsman, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Michal Segalov.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, p. 3, 2009.

Advances in Program Termination and Liveness

Byron Cook

Microsoft Research Cambridge

Abstract. Recent research advances now allow us to automatically prove termination and other liveness properties of many industrial programs. In cases where the desired property does not hold for all inputs, tools can be used to synthesize a precondition on the inputs under which the property does hold. In this tutorial I will describe these recent advances and discuss our efforts to apply termination analysis to industrial software.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, p. 4, 2009.

Verification of Security Protocols*

Véronique Cortier

LORIA, CNRS, Nancy, France

1 Introduction

Security protocols are short programs aiming at securing communications over a network. They are widely used in our everyday life. They may achieve various goals depending on the application: confidentiality, authenticity, privacy, anonymity, fairness, etc. Their verification using symbolic models has shown its interest for detecting attacks and proving security properties. A famous example is the Needham-Schroeder protocol [23] on which G. Lowe discovered a flaw 17 years after its publication [20]. Secrecy preservation has been proved to be co-NP-complete for a bounded number of sessions [24], and decidable for an unbounded number of sessions under some additional restrictions (*e.g.* [3,12,13,25]). Many tools have also been developed to automatically verify cryptographic protocols like [8,21].

In this tutorial, we first overview several techniques used for symbolically verifying security protocols that have led to the design of many efficient and useful tools. However, the guarantees that symbolic approaches offer have been quite unclear compared to the computational approach that considers issues of complexity and probability. This later approach captures a strong notion of security, guaranteed against all probabilistic polynomial-time attacks. In a second part of the tutorial, we present recent results that aim at obtaining the best of both worlds: fully automated proofs and strong, clear security guarantees. The approach consists in proving that symbolic models are *sound* with respect to computational ones, that is, that any potential attack is indeed captured at the symbolic level.

2 Symbolic Approach

In symbolic models, the implementation details of the primitives are abstracted away, and the execution is modeled only symbolically. The central characteristics of the symbolic approach is that messages are modeled using a term algebra T^f. Typically, the messages exchanged by parties are symbolic terms constructed from symbols for nonces n, identities a, by applying abstract operations representing encryption $\mathsf{enc}(m,k)$, pairing $\langle m_1, m_2\rangle$, signature $\mathsf{sign}(m,k)$, etc. Specifically, we consider the *signature* $\mathcal{F} = \{\mathsf{enc}, \mathsf{enca}, \mathsf{sign}, \langle\,\rangle, \mathsf{pub}, \mathsf{priv}\}$ together with arities of the form $\mathsf{ar}(f) = 2$ for the four first symbols and $\mathsf{ar}(f) = 1$ for the two last ones. The symbol $\langle\,\rangle$ represents the pairing function. The terms $\mathsf{enc}(m,k)$

* This work has been partially supported by the ARA project AVOTÉ and the ARA SSIA FormaCrypt.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 5–13, 2009.

and $\mathsf{enca}(m, k)$ represent respectively the message m encrypted with the symmetric (resp. asymmetric) key k. The term $\mathsf{sign}(m, k)$ represents the message m signed by the key k. The terms $\mathsf{pub}(a)$ and $\mathsf{priv}(a)$ represent respectively the public and private keys of an agent a. We fix an infinite set of *variables* $\mathcal{X} = \{x, y \ldots\}$. Terms with variables are used to specify the intended behavior of the protocol. The set T^f of *terms* is defined inductively by

$t ::=$		term
$\mid$	x	variable x
$\mid$	a	name a
$\mid$	$f(a)$	application of symbol $f \in \{\mathsf{pub}, \mathsf{priv}\}$ on a name
$\mid$	$f(t_1, t_2)$	application of symbol $f \in \{\mathsf{enc}, \mathsf{enca}, \mathsf{sign}, \langle\,\rangle\}$

The set of variables occurring in a term t is denoted by $\mathcal{V}(t)$. The set of *subterms* of a term t is denoted by $\mathsf{st}(t)$.

2.1 Intruder Deduction

Public network are insecure. We assume that a potential attacker, also called intruder, can not only listen to the network but also intercept, block and send new messages. The way an intruder can learn and build new messages is typically modeled through a deduction system. We give an example of such a deduction system in Figure 1. It corresponds to the usual Dolev-Yao rules. The first line describes the *composition* rules, the two last lines the *decomposition* rules. Intuitively, these deduction rules say that an intruder can compose messages by pairing, encrypting and signing messages provided he has the corresponding keys and conversely, he can decompose messages by projecting or decrypting provided he has the decryption keys. For signatures, the intruder is also able to *verify* whether a signature $\mathsf{sign}(m, k)$ and a message m match (provided he has the verification key), but this does not give him any new message. That is why this capability is not represented in the deduction system. We also consider a rule

$$\frac{S \vdash \mathsf{sign}(x, \mathsf{priv}(y))}{S \vdash x}$$

that expresses that an intruder can retrieve the whole message from his signature. This property may or may not hold depending on the signature scheme.

A term u is *deducible* from a set of terms S, denoted by $S \vdash u$ if there exists a *proof i.e.* a tree such that the root is $S \vdash u$, the leaves are of the form $S \vdash v$ with $v \in S$ (*axiom* rule) and every intermediate node is an instance of one of the rules of the deduction system.

Example 1. The term $\langle k_1, k_2 \rangle$ is deducible from the set $S_1 = \{\mathsf{enc}(k_1, k_2), k_2\}$. Indeed, a proof corresponding to that fact that $S_1 \vdash \langle k_1, k_2 \rangle$ is:

$$\dfrac{\dfrac{S_1 \vdash \mathsf{enc}(k_1, k_2) \quad S_1 \vdash k_2}{S_1 \vdash k_1} \qquad S_1 \vdash k_2}{S_1 \vdash \langle k_1, k_2 \rangle}$$

$$\frac{S \vdash x \quad S \vdash y}{S \vdash \langle x, y \rangle} \qquad \frac{S \vdash x \quad S \vdash y}{S \vdash \mathsf{enc}(x, y)} \qquad \frac{S \vdash x \quad S \vdash y}{S \vdash \mathsf{enca}(x, y)} \qquad \frac{S \vdash x \quad S \vdash y}{S \vdash \mathsf{sign}(x, y)}$$

$$\frac{S \vdash \langle x, y \rangle}{S \vdash x} \qquad \frac{S \vdash \langle x, y \rangle}{S \vdash y} \qquad \frac{S \vdash \mathsf{enc}(x, y) \quad S \vdash y}{S \vdash x}$$

$$\frac{S \vdash \mathsf{enca}(x, \mathsf{priv}(y)) \quad S \vdash \mathsf{pub}(y)}{S \vdash x} \qquad \frac{S \vdash \mathsf{sign}(x, \mathsf{priv}(y))}{S \vdash x}$$

Fig. 1. Intruder Deduction System

2.2 Protocols – Bounded Number of Sessions

Consider the famous Needham-Schroeder asymmetric key authentication protocol [23] designed for mutual authentication.

$$\begin{array}{ll} A \to B: & \mathsf{enca}(\langle N_A, A \rangle, \mathsf{pub}(B)) \\ B \to A: & \mathsf{enca}(\langle N_A, N_B \rangle, \mathsf{pub}(A)) \\ A \to B: & \mathsf{enca}(N_B, \mathsf{pub}(B)) \end{array}$$

The agent A sends to B his name and a fresh nonce (a randomly generated value) encrypted with the public key of B. The agent B answers by copying A's nonce and adds a fresh nonce N_B, encrypted by A's public key. The agent A acknowledges by forwarding B's nonce encrypted by B's public key. This protocol defines two *roles* that specify the behavior of the initiator agent A and the behavior of the responder agent B. Formally, the executions of a protocol are specified using terms with variables. We consider for simplicity a scenario where A starts a session with a corrupted agent I (whose private key is known to the intruder) and B is willing to answer to A. The initial knowledge of the intruder is

$$T_0 = \{a, b, i, \mathsf{pub}(a), \mathsf{pub}(b), \mathsf{pub}(i), \mathsf{priv}(i)\}.$$

The set $T_1 \stackrel{\mathsf{def}}{=} T_0 \cup \{\mathsf{enca}(\langle n_a, a \rangle, \mathsf{pub}(i))\}$ represents the messages known to the intruder once A has contacted the corrupted agent I. Then if a message of the form $\mathsf{enca}(\langle x, a \rangle, \mathsf{pub}(b))$ can be sent on the network, then B would answer to this message by $\mathsf{enca}(\langle x, n_b \rangle, \mathsf{pub}(a))$, in which case the knowledge of the intruder will turn to be

$$T_2 \stackrel{\mathsf{def}}{=} T_1 \cup \{\mathsf{enca}(\langle x, n_b \rangle, \mathsf{pub}(a))\}.$$

Subsequently, if a message of the form $\mathsf{enca}(\langle n_a, y \rangle, \mathsf{pub}(a))$ can be sent on the network, then A would answer by $\mathsf{enca}(y, \mathsf{pub}(i))$ since A believes she is talking to I and the knowledge of the intruder would be represented by

$$T_3 \stackrel{\mathsf{def}}{=} T_2 \cup \{\mathsf{enca}(y, \mathsf{pub}(i))\}.$$

The run corresponds to an attack if (for example), the intruder is able to learn the nonce n_b.

$$
\begin{array}{lllll}
R_1 & C \wedge T \Vdash u & \rightsquigarrow & C & \text{if } T \cup \{x \mid (T' \Vdash x) \in C, T' \subsetneq T\} \vdash u \\
R_2 & C \wedge T \Vdash u & \rightsquigarrow_\sigma & C\sigma \wedge T\sigma \Vdash u\sigma & \text{if } \sigma = \mathsf{mgu}(t,u),\ t \in \mathsf{st}(T), \\
 & & & & \quad t \neq u,\ t,u \text{ not variables} \\
R_3 & C \wedge T \Vdash u & \rightsquigarrow_\sigma & C\sigma \wedge T\sigma \Vdash u\sigma & \text{if } \sigma = \mathsf{mgu}(t_1,t_2),\ t_1,t_2 \in \mathsf{st}(T), \\
 & & & & \quad t_1 \neq t_2,\ t_1,t_2 \text{ not variables} \\
R_4 & C \wedge T \Vdash u & \rightsquigarrow & \bot & \text{if } \mathcal{V}(T,u) = \emptyset \text{ and } T \not\vdash u \\
R_f & C \wedge T \Vdash f(u,v) & \rightsquigarrow & C \wedge T \Vdash u \wedge T \Vdash v & \text{for } f \in \{\langle\,\rangle, \mathsf{enc}, \mathsf{enca}, \mathsf{sign}\}
\end{array}
$$

Fig. 2. Simplification Rules

This execution can be formally modeled by a *constraint system.*

Definition 1. *A* constraint system C *is a finite set of expressions* $T_i \Vdash u_i$*, where* T_i *is a non empty set of terms and* u_i *is a term,* $1 \leq i \leq n$*, such that:*

- $T_i \subseteq T_{i+1}$*, for all* $1 \leq i \leq n-1$*;*
- *if* $x \in \mathcal{V}(T_i)$ *then* $\exists j < i$ *such that* $T_j = \min\{T \mid (T \Vdash u) \in C, x \in \mathcal{V}(u)\}$ *(for the inclusion relation) and* $T_j \subsetneq T_i$*.*

The first condition says that the intruder knowledge only increases. The second condition ensures that variables are always first introduced on the right-hand side of a constraint. This corresponds to the fact that the output of a protocol depends deterministically on its entry.

A solution of a constraint system C is a substitution θ such that $\forall (T \Vdash u) \in C$, $T\theta \vdash u\theta$ holds.

Continuing our example, the set constraint corresponding to our scenario is:

$$T_1 \stackrel{\mathsf{def}}{=} T_0 \cup \{\mathsf{enca}(\langle n_a, a\rangle, \mathsf{pub}(i))\} \Vdash \mathsf{enca}(\langle x, a\rangle, \mathsf{pub}(b)) \quad (1)$$

$$T_2 \stackrel{\mathsf{def}}{=} T_1 \cup \{\mathsf{enca}(\langle x, n_b\rangle, \mathsf{pub}(a))\} \Vdash \mathsf{enca}(\langle n_a, y\rangle, \mathsf{pub}(a)) \quad (2)$$

$$T_3 \stackrel{\mathsf{def}}{=} T_2 \cup \{\mathsf{enca}(y, \mathsf{pub}(i))\} \Vdash n_b \quad (3)$$

Checking properties like confidentiality is thus reduced to finding a solution to set constraints. In our running example, a solution to the above set constraints corresponds to an attack on the confidentiality of the nonce n_b.

A way to solve set constraints is to transform them step by step into simpler ones [22]. A variant of the transformation rules, proposed by Hubert Comon-Lundh, are displayed in Figure 2. Using transformation rules, it can be shown that solving set constraints is an NP-complete problem [16,24]. It should be noticed that solving set constraints corresponding to checking security for one particular scenario. Once the number of sessions is fixed, there is finitely many (polynomially guessable) number of scenarios. This yields NP-completeness of secrecy for protocols for a bounded number of sessions.

2.3 Unbounded Number of Sessions

In the general case - when the number of sessions is not fixed - even a simple property such as secrecy is undecidable [18]. A classical restriction is to consider

a bounded number of nonces, meaning that the same nonce may be reused in different sessions, while this does not hold in reality. Such an abstraction may lead to false attack but allows to *prove* security properties [8,9] for an unbounded number of sessions. Checking properties like secrecy remains undecidable [18] but it enables to represent protocol execution using Horn clauses.

Typically, we consider a predicate I that represents the intruder knowledge. For example, the initial knowledge of the intruder for the Needham-Schroeder protocol can be modeled by the set

$$\mathcal{C}_{I_0} = \{I(a), I(b), I(i), I(\mathsf{pub}(a)), I(\mathsf{pub}(b)), I(\mathsf{pub}(i)), I(\mathsf{priv}(i))\}.$$

Abstracting nonces by constants, an unbounded execution of the Needham-Schroeder protocol can be represented by the following set $\mathcal{C}_{\mathsf{NS}}$ of clauses:

$$\begin{aligned} &\Rightarrow I(\mathsf{enca}(\langle n_a, a\rangle, \mathsf{pub}(i))) \\ I(\mathsf{enca}(\langle x, a\rangle, \mathsf{pub}(b))) &\Rightarrow I(\mathsf{enca}(\langle x, n_b\rangle, \mathsf{pub}(a))) \\ I(\mathsf{enca}(\langle n_a, y\rangle, \mathsf{pub}(a))) &\Rightarrow I(\mathsf{enca}(y, \mathsf{pub}(i))) \end{aligned}$$

For simplicity, we have only described the clauses corresponding to the case where A starts sessions with a corrupted agent I and B is willing to answer to A. To have a complete description of the protocol, one should also consider the case where A is willing to talk to B, B is willing to talk to I and symmetrically, all cases where A plays the role of B and B plays the role of A.

The ability of the intruder to analyze and forge new messages can be represented by the following set of clauses $\mathcal{C}_I$. It is the simple translation of the deduction system of Figure 1.

$$\begin{aligned} I(x), I(y) &\Rightarrow I(\langle x, y\rangle) & I(x), I(y) &\Rightarrow I(\mathsf{sign}(x, y)) \\ I(x), I(y) &\Rightarrow I(\mathsf{enc}(x, y)) & I(x), I(y) &\Rightarrow I(\mathsf{enca}(x, y)) \\ I(\langle x, y\rangle) &\Rightarrow I(x) & I(\langle x, y\rangle) &\Rightarrow I(y) \\ I(\mathsf{enc}(x, y)), I(y) &\Rightarrow I(x) & I(\mathsf{enc}(x, \mathsf{pub}(y))), I(\mathsf{priv}(y)) &\Rightarrow I(x) \\ I(\mathsf{enc}(x, \mathsf{priv}(y))) &\Rightarrow I(x) \end{aligned}$$

Then security of a protocol is reduced to checking satisfiability of a set of clauses. For example, the confidentiality of the nonce N_b can be expressed by the satisfiability of the set of clauses $\mathcal{C}_{\mathsf{NS}} \cup \mathcal{C}_I \cup \{\neg I(n_b)\}$.

This modeling of protocols is the approach used for by the ProVerif tool [8,10], which has been successfully used for analyzing many security protocols (see e.g. [1,11]). Some decidable fragments of Horn clauses, well suited for protocols have been proposed in [13,25].

3 Computational Approach

The abstraction of messages by terms and the limited adversary raise some questions regarding the security guarantees offered by such proofs, especially from the perspective of the computational model.

3.1 Brief Presentation of Computational Models

In computational models, messages that are exchanged are bit-strings and depend on a security parameter η which is used, for example to determine the length of random nonces. In contrast to symbolic models, the attacker does not perform predetermined actions for analyzing messages, but is modeled by any probabilistic Turing machine running in polynomial-time w.r.t. the security parameter.

Security properties are also stated in a stronger way than in symbolic models. For example, the confidentiality of a nonce does not only say that an attacker should not be able to output the nonce but also require that the attacker should not be able to get any partial information about the nonce. Formally, confidentiality is expressed through a game. The game is parametrized by a bit b and involves an adversary $\mathcal{A}$. The input to the game is the security parameter η. It starts by generating two random nonces n_0 and n_1. Then the adversary $\mathcal{A}$ starts interacting with the protocol Π. It generates new sessions, sends messages and receives messages to and from these sessions (as prescribed by the protocol). At some point in the execution the adversary initiates a session and declares this session under attack. Then, in this session, the confidential nonce is instantiated with n_b (*i.e.* one of the two nonces chosen in the beginning of the experiment, the selection being made according to the bit b) and the adversary continues its interaction with the protocol. In the end, the adversary is given n_0 and n_1 and outputs a guess d. The nonce is computationally secret in Π if the probability that $d = b$ is the same than the probability that $d \neq b$ up to some negligible function[1] in the security parameter.

Under the computational approach, the security of protocols is based on the security of the underlying primitives, which in turn is proved assuming the hardness of solving various computational tasks such as factoring or taking discrete logarithms. The main tools used for proofs are *reductions*: to prove a protocol secure one shows that a successful adversary against the protocol can be efficiently transformed into an adversary against some primitive used in its construction. Here, quantification is universal over *all* possible probabilistic polynomial-time (probabilistic polynomial time) adversaries and the execution model that is analyzed is specified down to the bit-string level. Two important implications stem from these features: proofs in the computational model imply strong guarantees (security holds in the presence of an *arbitrary* probabilistic polynomial-time adversary). At the same time however, security reductions for even moderately-sized protocols become extremely long, difficult, and tedious.

3.2 Bridging the Gap between Symbolic and Computational Models

Recently, a significant research effort aims at linking the two approaches via *computational soundness* theorems for symbolic analysis [2,4,5,6,17]. Justifying

[1] A function f is said to be negligible if it grows slower than the inverse of any polynomial, that is, for any polynomial P, there exists n_0 such that for any $n \geq n_0$, $|f(n)| \leq \frac{1}{P(n)}$.

symbolic proofs with respect to standard computational models has significant benefits: protocols can be analyzed and proved secure using the simpler, automated methods specific to the symbolic approach, yet the security guarantees are with respect to the more comprehensive computational model.

For example, it has been shown in [15] that for protocols with asymmetric encryption and signatures, any trace execution obtained by the interaction of a concrete (computational) adversary is (with overwhelming probability) the image of a symbolic execution trace obtained by executing a symbolic adversary. It is also proved that symbolic secrecy implies the computational (indistinguishability based) secrecy. This statement holds under standard assumptions on the security of the cryptographic primitives used in the concrete implementation, namely provided that encryption is IND-CCA2 secure and that signature is existentially unforgeable. Intuitively, IND-CCA2 security is defined through a game where the adversary should not be able to link cypher-texts to corresponding plain-texts even if he is given access to encryption and decryption oracles. A signature is existentially unforgeable whenever an adversary cannot produce valid signatures even being given access to a signature oracle.

One consequence of this result is that in the concrete model, the actions of an adversary are limited to the actions of the symbolic adversary. This allows to transfer trace-based security properties such as authentication and secrecy properties from symbolic to computational models. In other words, as soon as a protocol is proved secure for an authentication and secrecy property in symbolic models (using e.g. an automatic tool) then it is deemed secure in the less abstract computational model. This kind of results have then been extended e.g. to hash functions (in the random oracle model) [14], non-malleable commitment [19] and zero-knowledge proofs [7].

4 Conclusion

Symbolic approaches have proved their usefulness for analyzing security protocols. Automatic tools have been often used for discovering previously unknown flaws. Abstracting messages by terms seems to be a good level of abstraction since it is possible to show that security proof in symbolic models actually implies stronger guarantees in computational models under classical assumptions under the implementation of the primitives.

There are still several open directions of research. Symbolic approaches currently allow to check classical security properties such as confidentiality and authentication. For more recent protocols such as e-voting protocols and contract-signing protocols, the properties that should be achieved are more involved and cannot be encoded in existing tools. In addition, these recent protocols make use of less classical primitives such as re-randomizable encryption scheme or blind signatures. New decision techniques have to be developed for these particular primitives and security properties.

Bridging the gap between symbolic and computation models is a promising line a research since it enables to prove strong security guarantees, benefiting

from the simplicity of symbolic models. However, current results require strong assumptions on the security of the cryptographic primitives (e.g. IND-CCA2 encryption schemes). Weaker security assumptions like IND-CPA secure encryption schemes may not suffice to ensure security of protocols [26]. Using weaker encryption schemes may thus require to adapt both symbolic models and protocols accordingly.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just fast keying in the pi calculus. ACM Transactions on Information and System Security (TISSEC) 10(3), 1–59 (2007)
2. Abadi, M., Rogaway, P.: Reconciling two views of cryptography (the computational soundness of formal encryption). Journal of Cryptology 2, 103–127 (2002)
3. Amadio, R., Charatonik, W.: On name generation and set-based analysis in the dolev-yao model. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 499–514. Springer, Heidelberg (2002)
4. Backes, M., Pfitzmann, B.: Limits of the cryptographic realization of dolev-yao-style xor. In: de Capitanidi Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 336–354. Springer, Heidelberg (2005)
5. Backes, M., Pfitzmann, B.: Relating cryptographic und symbolic key secrecy. In: 26th IEEE Symposium on Security and Privacy, Oakland, CA, pp. 171–182 (2005)
6. Backes, M., Pfitzmann, B., Waidner, M.: A composable cryptographic library with nested operations (extended abstract). In: Proc. of 10th ACM Conference on Computer and Communications Security (CCS 2005), pp. 220–230 (2003)
7. Backes, M., Unruh, D.: Computational soundness of symbolic zero-knowledge proofs against active attackers. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008), Pittsburgh, PA, USA, June 2008, pp. 255–269. IEEE Computer Society Press, Los Alamitos (2008)
8. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proc. of the 14th Computer Security Foundations Workshop (CSFW 2001). IEEE Computer Society Press, Los Alamitos (2001)
9. Blanchet, B.: From secrecy to authenticity in security protocols. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 242–259. Springer, Heidelberg (2002)
10. Blanchet, B.: An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, Springer, Heidelberg (2005)
11. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: IEEE Symposium on Security and Privacy, Oakland, CA, May 2008, pp. 417–431. IEEE Computer Society Press, Los Alamitos (2008)
12. Blanchet, B., Podelski, A.: Verification of cryptographic protocols: Tagging enforces termination. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 136–152. Springer, Heidelberg (2003)
13. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 148–164. Springer, Heidelberg (2003)
14. Cortier, V., Kremer, S., Küsters, R., Warinschi, B.: Computationally sound symbolic secrecy in the presence of hash functions. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 176–187. Springer, Heidelberg (2006)

15. Cortier, V., Warinschi, B.: Computationally Sound, Automated Proofs for Security Protocols. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 157–171. Springer, Heidelberg (2005)
16. Cortier, V., Zalinescu, E.: Deciding key cycles for security protocols. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS, vol. 4246, pp. 317–331. Springer, Heidelberg (2006)
17. Datta, A., Derek, A., Mitchell, J.C., Shmatikov, V., Turuani, M.: Probabilistic Polynomial-time Semantics for a Protocol Security Logic. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 16–29. Springer, Heidelberg (2005)
18. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Undecidability of bounded security protocols. In: Proc. of the Workshop on Formal Methods and Security Protocols (1999)
19. Galindo, D., Garcia, F.D., van Rossum, P.: Computational soundness of non-malleable commitments. In: Chen, L., Mu, Y., Susilo, W. (eds.) ISPEC 2008. LNCS, vol. 4991, pp. 361–376. Springer, Heidelberg (2008)
20. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
21. Lowe, G.: Casper: A compiler for the analysis of security protocols. In: Proc. of 10th Computer Security Foundations Workshop (CSFW 1997), Rockport, Massachusetts, USA. IEEE Computer Society Press, Los Alamitos (1997); Also in Journal of Computer Security 6, 53–84 (1998)
22. Millen, J., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: Proc. of the 8th ACM Conference on Computer and Communications Security (CCS 2001) (2001)
23. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. Communication of the ACM 21(12), 993–999 (1978)
24. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions and composed keys is NP-complete. Theoretical Computer Science 299, 451–475 (2003)
25. Seidl, H., Verma, K.N.: Flat and one-variable clauses: Complexity of verifying cryptographic protocols with single blind copying. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 79–94. Springer, Heidelberg (2005)
26. Warinschi, B.: A computational analysis of the Needham-Schroeder protocol. In: Proc. 16th IEEE Computer Science Foundations Workshop (CSFW 2003), pp. 248–262 (2003)

Towards Automatic Stability Analysis for Rely-Guarantee Proofs

Hasan Amjad and Richard Bornat

Middlesex University School of Computing Science, London NW4 4BT, UK
Hasan.Amjad@cl.cam.ac.uk, R.Bornat@mdx.ac.uk

Abstract. The Rely-Guarantee approach is a well-known compositional method for proving Hoare logic properties of concurrent programs. In this approach, predicates in the proof must be proved invariant (or stable) under interference from the environment. We describe a framework, and a prototype implementation, for automatically detecting and repairing instability in such proofs. The method uses a combination of model checking, abstract interpretation, SMT and flow-control refinement.

1 Introduction

Multi-core and multi-processor computing systems are now mainstream. Hence, shared-memory concurrency, where multiple threads have read/write access to the same memory space, is becoming increasingly common. Consequently, concurrent programs are the focus of much recent research on automatically proving program properties. Often, the assertions we would like to prove are not amenable to existing automatic analyses. This paper studies one such scenario, and shows how existing automatic techniques can nonetheless help the proof process.

The main challenge in proving properties of concurrent programs, and indeed in their design, is dealing with interference, i.e., the possibility that threads may concurrently make changes to the same memory address. The concurrent programming community has evolved several synchronisation schemes to avoid interference. Most rely on some form of access denial, such as locks. Though locks make it easy to reason about correctness, they may also cause loss of efficiency as threads wait to acquire access to needed resources. Locking schemes have thus become increasingly fine-grained, attempting to deny access to the smallest possible size of resource, to minimise waiting and maximise concurrency. The ultimate form of such fine-grained concurrency are programs that manage without any synchronisation at all [17].

The finer the concurrency, the more involved the logic for avoiding interference. This logic must implicitly or explicitly take the actions of other threads into account. This is a problem for program proofs where we strive for modularity, i.e., we wish to be able to reason about a piece of code in isolation from the various environments in which it could execute.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 14–28, 2009.

Rely-guarantee (RG) reasoning [16] offers a fairly widely used solution to this problem within the framework of Hoare-style program proofs [14]. Broadly speaking, RG encodes the environment into the proof: all pre- and post-conditions of every Hoare triple must be shown to be unaffected (or stable) under the actions of the environment. Once non-interference has been established, the proof can be carried forward exactly as for a sequential program. Automatically ensuring such non-interference can be problematic in many cases.

In this paper, we describe preliminary progress on a possible solution, that relies critically on state-of-the-art tools from the model checking, abstract interpretation and Satisfiability-modulo-Theories (SMT) research communities. Our stability analysis engine can be integrated with a verification condition generator, allowing greater automation for Hoare logic proofs of concurrent programs.

The next section gives relevant background. We then describe our method, and give examples to illustrate shortcomings and possible developments. We assume some familiarity with program proofs in Hoare logic [14], and with model checking [7].

2 Preliminaries

2.1 Rely-Guarantee Reasoning

Rely-guarantee (RG) is a compositional verification method for shared memory concurrency introduced by Jones [16]. Interference between threads is described using binary relations. In that treatment, post-conditions were relational, so assertions could talk about the state before and after an action. Here, in line with traditional Hoare logic, we shall use post-conditions of a single state, as this usually makes for simpler proofs. In either case, the essence of RG is unaffected by this choice.

Our command language will be the one used by Jones, i.e., with assignment, looping, branching, sequential composition and parallel composition, using C-like syntax. For parallel composition we assume standard interleaved execution semantics, i.e., the threads of a program have access to some shared state, and atomic instructions occur interleaved.

Program variables will range over $\mathbb{N}$. It may seem odd to have program variables range over infinite types. In practice however, reasoning about numbers with the aid of abstraction, has been found to be more tractable than reasoning about finite but huge state spaces over machine words or bit-vectors, which are harder to abstract due to overflow and underflow corner cases. We made this choice primarily for ease of tool use while developing our prototype.

RG can be seen as a compositional version of the Owicki-Gries method [18]. The specification for a command C is a four-tuple (P, R, G, Q), where P and Q are the usual Hoare logic pre- and post-condition assertions on a single state, and R and G are binary relations on states (often called "two-store" predicates).

R and G summarise the properties of the individual atomic actions invoked by the environment and the thread respectively. An action is given as a binary relation on the shared state, and is written $g \rightsquigarrow \bar{v} := \bar{e}$. This notation indicates

that, if the guard g is true, then the action simultaneously updates the vector of shared state variables $\bar{v}$ pointwise with the value of the expression vector $\bar{e}$, and does nothing otherwise. Note that that update need not be atomic. For example, the action corresponding to the command `x = x + 1`, that increments a shared integer x, might be written as $\mathbf{true} \rightsquigarrow x := x + 1$. We will elide **true** guards.

C satisfies its specification, $(P, R, G, Q) \vDash C$, if from a state satisfying P, and under environmental interference at most R (the *rely*), C causes interference at most G (the *guarantee*), and if it terminates, it does so in a state satisfying Q.

G is the relation given by the reflexive and transitive closure of all actions of the thread being specified. R is similarly the reflexive and transitive closure of all the actions of the environment, i.e., the actions comprising R are just the G actions of all the other threads. We overload the notation for R and G to indicate either relations or predicates, as convenient.

Taking reflexive and transitive closures has the effect of forgetting all control-flow information. This has the disadvantage of overapproximating the behaviours of the environment, but the advantage that any proof will hold for arbitrary numbers of threads of the proved code (new code may of course break the proof).

The actions are given by manual annotation, as in general, automatic action discovery is non-trivial. The reason is that actions can only mention shared state variables, whereas in the code shared state variables may often be assigned the value of an expression containing non-shared (or *local*) variables. Fully automatic action annotation would thus require an iterative computation that may end up exploring the full state space of the program, and this we wish to avoid. For the toy programs we have looked at so far, this has not been a problem. For very large programs, one can always annotate a procedure call with a single action, and not descend into the call itself.

An assertion P on a single state is considered *stable* under interference from a binary relation R if $(P; R) \Rightarrow P$, i.e., if $P(s)$ and $(s, s') \in R$, then $P(s')$. More specifically, if P is the pre-condition for some command C, then it must continue to hold after any environment action, before the execution of C.

Jones gives a full proof system for the satisfaction relation, but we will not need it for this work. However, we reproduce the two critical rules here, to make our assumptions about RG concrete. The first rule is parallel composition, where $||$ is the interleaving parallel composition operator.

$$\frac{(P_1, R \vee G_2, G_1, Q_1) \vDash C_1 \qquad (P_2, R \vee G_1, G_2, Q_2) \vDash C_2}{(P_1 \wedge P_2, R, G_1 \vee G_2, Q_1 \wedge Q_2) \vDash C_1 || C_2} \tag{1}$$

For our purposes, we assume that a read or write of a single boolean or integer is the highest level of hardware atomicity, e.g., writing two booleans is not hardware atomic. This assumption is satisfied by most machine architectures. The second rule tells us what it means for a command to be atomic,

$$\frac{(P, \mathrm{id}, G, Q) \vDash C \qquad P \text{ stable under } R}{(P, R, G, Q) \vDash \mathit{atomic}(C)} \tag{2}$$

meaning that if a command satisfies a specification with an empty rely, and the pre-condition is stable under some rely R, then the command may be treated as if it executed atomically. Hardware atomic commands are of course trivally atomic.

Note one departure from standard RG: the post-condition of the very last line of code is not checked for stability. It is instantaneously true immediately after execution of that line. At this point, either the thread terminates, so that we do not care whether the environment interferes with the post-condition, or, the thread resumes execution from some command the pre-condition of which will be the same as this post-condition, and thus will be checked for stability.

2.2 Temporal Logic Model Checking

Let V be the set of program variables (or *state variables*) used in a program (with appropriate scope management, which we ignore without loss of generality). Each $v \in V$ ranges over a non-empty set of values D_v. The state space S of the program is given by $\prod_{v \in V} D_v$. A single state of the program is then a value assignment to each $v \in V$.

Suppose AP is the set of all those atomic propositions over V that we might use in the specification of a program. Then we can turn the program into a state machine M represented as a tuple $(S, S0, T, L)$ where S is the set of states, $S0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and $L : S \to 2^{AP}$ labels each state with the subset of AP that is true in that state.

A temporal logic augments propositional logic with modal and fix-point operators. The semantics of a temporal logic formula in which the atomic propositions range over AP can be expressed in terms of sets of states and/or sequences of states of M. If we turn a program into a state machine, we can use temporal logics to express temporal properties of the program.

The most common such property is the global invariant, i.e., a property that holds in all reachable states of a state machine, or equivalently, always holds during the execution of a program. We denote this by

$$M \vDash \mathbf{G} f$$

where M is the state machine, f is the global invariant and $\mathbf{G}$ is the "Globally" temporal logic operator. The semantics of Gf are simply that f must hold in every state of M that is reachable from the initial states of M.

$M \vDash \mathbf{G} f$ can be checked automatically using proof procedures known as model checkers, subject only to time and space constraints. More importantly, if the proof attempt fails, the model checker can return a counterexample, which is an execution path (sequence of transitions) leading from an initial state to a state in which the invariant is not satisfied. See [7] for details.

The problem of model checking global invariants is in general undecidable when the state space is infinite. However, the ability to produce counterexamples has led to the development of counterexample guided abstraction refinement (CEGAR) [8,19], where the state space is first abstracted to a simpler one,

and if the constructed abstraction is too general it can often be automatically iteratively refined until the desired property is verified.

We do not need to describe model checking or CEGAR in more depth, particularly as there are many different abstraction schemes and CEGAR techniques. Further details may be found in [7,8,13,19].

3 The Problem of Instability Detection and Repair

Rule (2) of RG is used to establish a command as atomic. The command is then free from interference, and can be reasoned about using standard Hoare logic rules for sequential programs. The key requirement is to prove that the precondition P of the command is stable under the rely R. This is a two-step process. First, one *detects* whether or not P is stable. Second, if P is unstable, one *repairs* it; the usual way is by weakening P. This stability analysis is thus the key to reducing proofs about concurrent programs to proofs about sequential programs.

Instability can be detected either manually, or sometimes automatically using syntactic checks [22]. Once detected, it is repaired by weakening P, either manually or using ad hoc abstraction heuristics if the underlying domain permits [6,9].

As a simple example, consider the assertion $x = 10$ that is unstable under the environment consisting of the single action $x := x + 1$. In general, detecting this automatically is not straightforward.

To repair this, a common approach is to disjunctively add state based on the action, starting with the states satisfying the assertion, i.e.,

$$\begin{array}{l} (x = 10) \\ (x = 10 \vee x = 11) \\ \vdots \end{array}$$

and hope to reach a fixpoint. Unfortunately, in this case such naive automatic stabilisation will not terminate. To repair manually, we use the boolean abstraction $\alpha(x) \iff x \geq 10$, where α is a total *abstraction function* [7]. Under this abstraction the action above becomes the identity action (now on booleans) in all cases except when $x = 9$, but in that case $x = 10$ does not hold anyway, so we have stability immediately, and the assertion is stabilised to $x \geq 10$.

Automatic instability detection and repair are thus both non-trivial problems. Further, we observe that weakening P is not the only way to achieve stability. From the definition of stability, strengthening the rely is also a possible solution. We shall present a method that combines both approaches.

We have seen in §2.2 that this problem of finding exactly the right level of abstraction also occurs in model checking. It is our hope that the model checking solution (i.e., CEGAR) can be applied to stability analysis as well. If so, the vast amount of model checking research on this topic can be brought to bear on the problem.

4 Automatic Stability Analysis

Our solution is to represent R and G as state machines M_R and M_G of actions. Stability detection for a predicate P is then reduced to model checking that P is a global invariant over M_R. If the check fails, we use the counterexample trace provided by the model checker to either strengthen R, or weaken P, and then repeat the check until success or irreparable failure. The process can diverge because over an infinite state space there may be an infinite sequence of successively weaker assertions, none of which is stable. For this reason, for now we always prefer strengthing. A further novelty of our approach is that we perform weakening only with respect to the part of the environment that is causing instability (rather than the full environment), thus reducing the risk that the program proof will fail because the stabilised precondition was too weak.

4.1 Detection

The state machine for the guarantee condition G_t for a thread t, is $M_t = (S_t, S0_t, T_t, L_t)$ and is constructed as follows. Let V_t be the set of all shared program variables used in t. Then S_t is constructed like S in §2.2. $S0_t$ is those states of S_t in which all $v \in V_t$ are assigned their initial values if any. Let a_l be the (possibly empty) set of actions associated with the command on line l of the thread code (having a set of actions associated with a command allows us to abstract procedure calls, and also to use auxilliary actions without violating atomicity constraints). Then

$$T_t(s, s') = \bigvee_l \left(\bigwedge_{a \in a_l} a(s, s') \right)$$

Note that this transition relation is not tracking control-flow. Finally, $L_t(s) = \{p \in AP \mid p(s) = \mathbf{true}\}$. L_t is a technical requirement for defining the semantics of temporal logics; it gives the set of states in which a given atomic proposition $p \in AP$ is true.

Now suppose we have the guarantee state machines for all threads of the program. To do stability detection for some assertion P in the proof of some thread, let the environment E be the set of all other threads (i.e., all threads less the one for which the proof is being done). Then we construct the state machine for R, $M_R = (S_R, S0_R, T_R, L_R)$ over variables $V_R = \bigcup_{t \in E} V_t$, as follows

$$\left(\prod_{v \in v_R} D_v, \prod_{v \in V_R} init(v), \biguplus_{t \in E} T_t, \lambda s.\{p \in AP \mid p(s) = \mathbf{true}\} \right)$$

where $init(v)$ gives the set of possible initial values of v. Note that the transition relation T_R is simply a union of the transition relations of the environment's threads, so it is also not tracking control flow.

Concretely, the transition system of M_R non-deterministically chooses a thread $t \in E$, after which M_R behaves like M_t for the duration of the execution

of some action of t, after which it again picks a thread and repeats. Once we have M_R, the stability check is reduced to invoking a model checker to confirm that

$$M_R \vDash \mathbf{G}P$$

Theorem 1. *Ignoring valuations of P on unreachable states of M_R, we have*

$$M_R \vDash \mathbf{G}P \Rightarrow (P; R) \Rightarrow P$$

Proof. If $M_R \vDash \mathbf{G}P$ then P holds in all reachable states of M_R. A state s is reachable iff for $s_0 \in S0_R$ $R(s_0, s)$. Now P holds in all states, so in particular in $s_0 \in S0_R$, and all s and s' such that $R(s', s)$. Since $(P; R) \Rightarrow P$ is equivalent to $\forall ss'.P(s) \wedge R(s, s') \Rightarrow P(s')$, we have our result. □

Note that ignoring valuations of P on unreachable states of M_R is fine because we do not care whether P is stable or unstable for situations that can never happen. It is possible that P may be stable on all reachable states and unstable only in some unreachable states, in which case the model checker may still fail the stability check (since the reachable states of M_R over-approximate the reachable states of the environment). This is a completeness issue, and is dealt with by a refinement-based solution in §4.2.

So if P is a global invariant of M_R then P is stable under R. If not, the model checker will supply a counterexample, which will be a sequence of actions leading to a state in which P is not satisified.

A global invariant is a safety property in the sense that a violation of the property is caused by the explicit occurrence of an observable event. We note here that rule (1) of RG, which appears to be performing circular reasoning, is sound for safety properties [1].

We use the software model checker BLAST [5], to check that $M_R \vDash \mathbf{G}P$. Forgetting control flow considerably simplifies M_R, easing the model checker's task. Our prototype implementation generates a C program corresponding to M_R, annotated with a BLAST assertion corresponding to P. The non-deterministic thread selection is easily constructed from the BLAST boolean non-deterministic choice primitive. We denote by $mc(M, P)$ a call to the model checker, that checks whether assertion P is a global invariant over state machine M. This returns either **true** or a counterexample trace π as a sequence of actions.

Indirectly, the stability check relies on the model checker's underlying SMT solver's ability to check assertions in combinations of various theories. Thus, we can do instability detection in any theory supported by BLAST (or some software model checker), and so are able to always use state-of-the-art software model checking techniques for free. At this point we are already ahead of the game by not being limited to requiring syntactic stability checks.

4.2 Repair

As mentioned earlier, the standard approach to instability repair is by weakening P, but an alternative is to strengthen R. The latter approach can be useful if

weakening P would make it impossible to prove the Hoare logic property we are interested in. Also, as noted in §4.2, too weak an R can lead to incompleteness in stability detection. We now present a method that combines both approaches.

In standard RG, R and G are the reflexive transitive closures of their constituent actions. This representation corresponds to state machines that can perform any action from any state. Indeed, M_R above is precisely this state machine (since the transition relation is not tracking control-flow).

The obvious way to strengthen R is to re-introduce control-flow information into M_R. If the stability of P depends on the sequencing of actions, then such a refinement of R will allow us to stabilise P without weakening it. We shall see an example of this later. On the other hand, by introducing control-flow in R, we begin losing scalability in the sense that the model checker's task becomes harder, and the overall proof need not hold for arbitrary numbers of threads of the same piece of code, but only for the number that is model-checked.

To do this strengthening, we need to be able to track control-flow in each thread. Four preliminary steps are required:

1. We add, for each action a of each $t \in E$, the set of all actions of t that may execute immediately after a. We call this the *successor set* of a, denoted by $succ(a)$.
2. We attach to each action a a boolean flag $pin(a)$. If $pin(a)$ is true, then we say that a is *pinned*, i.e., intuitively, we are now tracking control flow for a.
3. For each thread t, we add a special dummy action $start_t$, which is never executed, but for which $succ(start_t)$ is the set of all actions of t that can be the very first action executed when t starts.
4. We maintain a function $last(t)$, which returns the most recent action executed by t, or $start_t$ if t has not executed any action yet.

Till now, after the non-deterministic choice of thread in T_R, the action to execute was chosen non-deterministically as well. Now, let $choose(n)$ for $n > 0$ be the function that non-deterministically chooses a number between 0 and $n-1$, and let $|t|$ be the number of distinct actions of a thread t. Suppose thread t is picked as the next thread to run. The choice of next action in M_R is then determined as follows:

procedure *pick_action*(t)
1. $tmp \leftarrow choose(|t|)$
2. **if** $(pin(tmp))$ **then**
3. **if** $tmp \notin succ(last(t))$ **then goto** 1
4. **return** tmp

The intuition is that an unpinned action can execute at any time, whereas a pinned action is only allowed to execute if it is in the successor set of the most recently executed action of that thread. This allows fine-grained control over adding execution sequencing information to M_R. This code is never executed, only model-checked, so non-determinism and infinite loops are perfectly acceptable.

The rely control-flow refinement then works as follows (recall that *mc* is a call to the model checker, return either **true** or a counterexample trace π):

procedure $mcr(M_R, P)$

1. $\forall a.pin(a) \leftarrow$ **false**
2. **match** $mc(M_R, P)$
3. **case true** : **return** SUCCESS
4. **case** π : let $\langle a_0, a_1, \ldots, a_m \rangle = \pi$
5. **if** $\forall a \in \pi.pin(a)$ **then return** FAIL
6. $tmp \leftarrow a_i \in \pi$ s.t. $\neg pin(a_i) \wedge \forall j > i.a_j \in \pi \Rightarrow pin(a_j)$
7. $pin(tmp) \leftarrow$ **true**
8. **goto** 2

The rely refinement loop works by adding control-flow information for actions in the counterexample trace until either the model checker reports success, or returns a counterexample trace in which we are already tracking control-flow for all actions of the trace. In the latter case, it is clear that there is at least one execution sequence where stability fails even when nothing executes out of sequence.

We must ensure that rule (1) of RG is still sound since the rely may now not be the reflexive transitive closure of the environment's actions. The other rules are not affected. The key is to ensure that the guarantee of each thread is stronger than the rely of every other thread. This is indeed the case: since our refinement only re-introduces control-flow information, the refined state machine will never under-approximate the actual program, and so the guarantees will allow all valid program executions.

With *mcr*, we simply fail if further control-flow refinement of the rely cannot repair instability. We now add weakening of P. For this, we amend *mcr* to return π instead of FAIL in line 5. Further, we assume access to a procedure $weaken(M)$ which infers invariants on variable occurrences at various control points in a given state machine, and returns the list of invariants thus found.

We use the INTERPROC tool [15] for this functionality, as it provides an easy to use interface to state-of-the-art abstract interpretation libraries for numerical domains. Since we are ignoring the heap for now, our case studies use mainly arithmetic and static arrays. The input of INTERPROC is a program in an academic imperative language that supports numerical types. The underlying engine of INTERPROC consists of a generalised fixpoint computation which in turn uses abstraction interpretation libraries such as APRON [3]. Since these work over numerical domains only, our current examples are also limited to assertions over number-valued variables.

The output is the program code annotated with invariants on program variable occurrences at various control points in the code. Invariants at the same control point hold simultaneously and can be conjoined. Those at distinct control points do not hold simultaneously and can be disjoined. Thus we can summarise the information from INTERPROC as a formula in disjunctive normal form.

We denote the call to INTERPROC and the invariant parser by $weaken(M)$, and add assertion weakening to our method as follows

procedure $mcrw(M_R, P)$
1. **match** $mcr(M_R, P)$
2. **case** SUCCESS : **return** SUCCESS
3. **case** π : let $\langle a_0, \ldots, a_m \rangle = \pi$
4. $dnf \leftarrow weaken(M_R \restriction \pi)$
5. **if** $(P \Rightarrow P \vee dnf) \wedge \neg(P \vee dnf \Rightarrow P)$
6. **then** $P \leftarrow P \vee dnf$
7. **else return** FAIL
8. **goto** 1

The call to *mcr* may statefully refine M_R. In line 4, $M_R \restriction \pi$ is the state machine that is exactly as M_R but restricted to the actions in π (so it automatically inherits any control-flow refinement). Line 5 uses the CVC3 SMT solver [4] to evaluate the test. If an assertion strictly weaker than P cannot be found, the stability repair attempt declares failure.

The call to *weaken* operates only on the part of the environment that is causing instability. This ensures that we do not weaken P unnecessarily, and also eases the abstract interpreter's task. The call to *mcr* resets all pins. This slows down convergence, but also decreases the likelihood of returning an unnecessarily strong R. As noted earlier, the weaker the R and the stronger the P, the better.

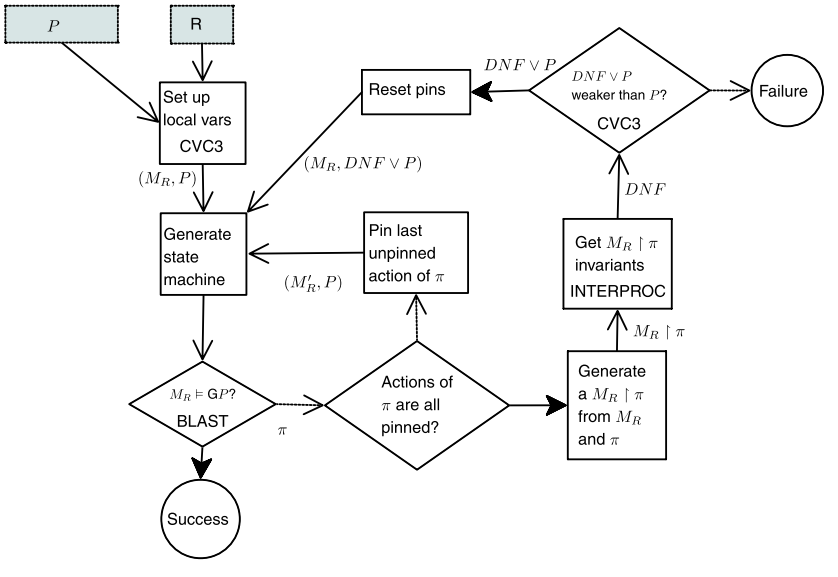


Fig. 1. Overview of Framework

An overview of the framework is given in Figure 1. Solid arrowheads indicate positive branches at decision points. Shaded boxes indicate manual work: first, the user must annotate code with actions, from which R can be automatically constructed (it is manual for now); second, the assertion to stabilise must be supplied.

5 Examples

We give a few simple examples to illustrate the framework and the limitations of our current implementation.

Example 1. Let the environment be

1. int $x = 1$
2. $x = x + 1$
3. $x = x - 1$
4. **goto** 2

and suppose we wish to check the stability of the stable assertion $P \equiv x \geq 1 \wedge x \leq 2$. In standard RG, the actions of this environment are $x := x + 1$ and $x := x - 1$ (the initialisation of x is absorbed into the initial state). P is not stable under R, which is the reflexive transitive closure of these actions, e.g., R could consecutively execute the increment twice.

Invoking the stability analysis, the model checker fails, and the counterexample shows either $x > 2$ because of the first action or $x < 1$ because of the second action. The offending action is then pinned, but the second run of the model checker will fail again, because the remaining unpinned action can again occur thrice consecutively. The analysis pins this action as well, and now the check succeeds because both actions must now alternate. This example also shows how our method can prove stronger properties than are possible in standard RG, despite being automatic.

Example 2. We now see a case which requires weakening for success. Suppose the environment is

1. int $x = 1$
2. $x = x + 1$
3. **goto** 2

and we wish to check the stability of the assertion $P \equiv x = 1$. This is unstable, because of the increment.

The model checker fails, and the counterexample points to $x := x+1$. Pinning this, the model checker fails again. Now the counterexample trace contains no unpinned actions, so the analysis concludes that there is at least one way to destabilise the assertion even with full control-flow tracking. It calls the abstract

interpretation engine for help, which in this case returns the invariants $x-1=0$ for line 1, and $x \geq 1$ for line 3. The analysis then uses an SMT solver to confirm that $P \vee (x-1=0 \vee x \geq 1)$ is strictly weaker than P, and weakens P to $x-1=0 \vee x >= 1$. The model checker now succeeds even with all pins reset.

This example illustrates the use of weakening, but also exposes one of the shortcomings of the method as currently implemented. There is a tradeoff between weakening P and strengthening R, and the current approach always prioritises the latter. We are currently investigating more sophisticated algorithms that attempt to efficiently find the strongest P and weakest R for which stability is provable.

Example 3. The above examples may have given the impression that we always end up with full control-flow tracking. But consider the environment

1. int $x = 10, y = 10$
2. $x = 10$
3. $y = 10$
4. **if** $x \leq y$ **then** $x = x + 1$
5. **if** $y \leq x$ **then** $y = y + 1$
6. **goto** 4

and let $P \equiv x - y \leq 1 \wedge y - x \leq 1$. The actions are

1. $x := 10$
2. $y := 10$
3. $x \leq y \rightsquigarrow x := x + 1$
4. $y \leq x \rightsquigarrow y := y + 1$

and P is not stable under the initial R (which does not track any control-flow), because the first two actions can execute any time. However, the order of the last two actions does not matter, and indeed the analysis does pin the first two only.

Example 4. We now consider a concrete example that occurred in our own research. During the construction of a correctness proof of Simpson's 4-slot algorithm [20], the second author devised the algorithm of Figure 2. This simulates a lock-free one-place buffer that is concurrently accessed by a single writer thread and a single reader thread. The implementation uses as shared state a 2-place array b and two integers l and c, and manages reads and writes without corruption and in sequence (but reader starvation is possible). Reads and writes are atomic at bit level only, so in particular the read and write of b are not atomic.

Safety (i.e., reader never returns a corrupt value) can be model-checked. Less obvious is *freshness*. This states that the reader must always read a value that is either the same as or occurs later in the sequence of written values, than the latest such value when the reader thread started.

```
write(x : T)          ||  read() returns y : T
  local t:int         ||    local t:int
  c = 1               ||    do c = 0
  t = 1 − l           ||      t = l
  b[t] = x            ||      y = b[t]
  l = t               ||      t = c
                      ||    until t == 0
                      ||    return y
```

Fig. 2. Lock-free one-place buffer

For freshness, we can assume without loss of generality that the values written to b are the serial numbers of the written values. Then we construct the environment from the user supplied writer actions (i.e., $c := 1$, $b[\neg l] := b[l] + 1$ and $l := \neg l$), supply the reader pre-condition $v = b[l]$ and the post-condition $y \geq v$, and do a standard strongest-postconditions forward analysis, using the algorithm for stability checking. Both the environment construction and forward analysis can be automated, and we plan to do so in due course.

We expect that automatically proving this can also be done with an infinite state model checker that can handle parallel composition. It would need manual code instrumentation with observer variables (effectively simulating serial numbers), since temporal logic cannot otherwise capture this property. However, this approach is unlikely to scale as thread sizes grow, whereas our approach completely sidesteps state space explosion in the thread being verified, and will only possibly encounter state space explosion in the environment if the cause of instability is unrelated to control-flow.

6 Related Work

We do not know of any other work that uses model checking and abstraction for stability analysis in Hoare-style RG proofs. There is work underway at MSR Cambridge [12] that also represents R and G as state machines, but their aim is to deal with questions of liveness. Other than that we know only of the work on automatic stabilisation for a fragment of separation logic [6,21], which uses weakening only and is restricted to syntactic checking for instability.

There has been work on modular verification using model checking, where the component being verified is distinguished from its environment (which can be abstracted and refined, e.g., [2]). Our work is both less powerful in that it requires the user to provide the environment (i.e., the rely) manually, and more powerful in that it is able to use Hoare logic, thus sidestepping the state explosion problem within the component being verified.

The work on thread-modular verification by Flanagan et al [11] is more relevant. They also use a rely-guarantee style approach, but in which executions of the rely are inserted throughout the code of the thread being verified, and at each step, it is checked that the guarantee of the thread holds. Their work is

more mature, with support for procedure calls and connections with sequential program checkers. However, their approach can be seen as a special case of our framework: the case where proof fails if it fails for the initial environment and specification.

7 Concluding Remarks

This paper only establishes proof-of-concept. For a start, we would like to construct the environment automatically from the annotations, so that we do not have do our proofs one thread at a time. To handle possible environment state-space explosion in the stability detection phase, we propose the use of abstract separation logic to carve out irrelevant state. Separation logic and RG have already been combined [22]. We may also use INTERPROC supplied invariants to help the model checker with abstraction refinement.

The algorithm has other important limitations. It may not terminate. There are proof scalability issues (see §4.2). It inherits from RG the limitation to parallel composition (no forks/joins), and to proving safety properties only. Further, we completely ignore the heap.

Nonethless, the current implementation is useful (e.g., §5), and, apart from possible non-termination, the limitations are not permanent [10,12].

Acknowledgement. The first author would like to thank Viktor Vafeiades for permission to reproduce excerpts from the description of RG in his Ph.D. thesis.

References

1. Abadi, M., Lamport, L.: Conjoining specifications. Technical Report 118, DEC Systems Research Center (1993)
2. Alur, R., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Automating modular verification. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 82–97. Springer, Heidelberg (1999)
3. APRON project, `http://apron.cri.ensmp.fr/`
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9(5-6), 505–525 (2007)
6. Calcagno, C., Parkinson, M.J., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
9. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)

10. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-guarantee reasoning (2008) (Draft)
11. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. 338(1-3), 153–183 (2005)
12. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving liveness properties of non-blocking data structures. In: POPL (submitted, 2008)
13. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
14. Hoare, C.A.R.: An axiomatic basis for programming. Communications of the ACM 12(10), 576–580 (1969)
15. Interproc static analyzer, `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
16. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
17. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC 1996: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, pp. 267–275. ACM Press, New York (1996)
18. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM 19(5), 279–285 (1976)
19. Saïdi, H.: Model checking guided abstraction and analysis. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 377–396. Springer, Heidelberg (2000)
20. Simpson, H.R.: Four-Slot Fully Asynchronous Communication Mechanism. IEE Proceedings 137(1), 17–30 (1990)
21. Viktor Vafeiadis. Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
22. Vafeiadis, V., Parkinson, M.J.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)

Mostly-Functional Behavior in Java Programs

William C. Benton and Charles N. Fischer

University of Wisconsin–Madison
1210 W. Dayton St. Madison, WI 53706

Abstract. We present a lightweight type-and-effect system for Java programs that features two major innovations over extant object-oriented effects systems: *initialization effects*, which are writes to an object's state while it is being constructed, and *quiescing fields*, which are fields that are never written after an object is constructed. We also present a novel taxonomy of *degrees of method purity* in object-oriented programs, which characterizes methods whose effects are confined to their receiver object. Finally, we find significant amounts of *mostly-functional* behavior in realistic Java programs: in the benchmarks we analyzed, between 48–53% of declared fields were identifiable as quiescing and between 24–78% of dynamic field reads were from quiescing fields.

Keywords: Java, Program Analysis, Type-and-effect Systems.

1 Introduction

Effect systems extend classical type systems with information about the computational effects exhibited by expressions, statements, and methods. Just as type signatures characterize the range of values an expression may assume, effect signatures can provide concise, useful summaries of the potential effects of a particular method invocation. Because of this capability, effect systems currently enjoy widespread application in several problem domains, including program analysis, semantics-preserving program transformation, software understanding, verification, and compile-time memory management.

In this paper, we present two innovations that can increase the expressivity and precision of effect signatures. *Initialization effects* are writes that occur to the state of an object while it is being constructed but before it is available to the rest of the program; *quiescing fields* are instance variables of an object whose values remain constant after its constructor returns. We present these in the context of a fairly simple effects system for Java, but these novel features are based on concepts orthogonal to the underlying effects system and could be adapted to more expressive systems.

We also present a notion of function purity that exploits the engineering properties of object-oriented programs: namely, that mutable state is typically accessed through the interface of the object that contains it. We describe instance methods that are pure with respect to all mutable state outside of their receiver

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 29–43, 2009.

object as *externally pure*; we refer to methods that may read (but not write) external mutable state as *externally read-only*.

Perhaps most surprisingly, we show that realistic Java programs exhibit a substantial degree of *mostly-functional* behavior. "Mostly-functional," as coined by Knight [1], describes a programming discipline in which the presence and extent of computational effects are limited as much as possible. In the context of Java, this includes both accesses to quiescing fields — which are read-only after the object is available to the rest of the program — and the prevalence of externally-pure and externally read-only methods, whose updates to mutable state are only visible via an object's interface.

These results have several consequences for program analysis, verification, and understanding. Annotations on (externally-) pure or read-only methods aid interprocedural dependence analysis, serve as part of a method's specification, and provide documentation to human programmers. Furthermore, some such methods may be amenable to aggressive code scheduling optimizations, including asynchronous execution on multicore processors.

1.1 Overview

In the remainder of this paper, we will introduce object-oriented effects systems (Section 3) and present the simple effects system and inference rules for Java that will form the basis for our subsequent developments (Section 3.2). After this preliminary discussion, we shall introduce our major contributions:

1. The concept of *initialization effects*, which are writes that occur to a field of an object while it is being constructed, and rules to infer these (Section 4);
2. The concept of *quiescing fields*, which are never written after their containing object is constructed, and a rule to infer such fields (Section 5); and
3. A novel taxonomy of *degrees of method purity*, which extend conventional definitions of function purity to account for methods whose effects are confined to their receiver object (Section 6).

We conclude by placing our work in the context of related research efforts (Section 7) and suggesting future investigations.

2 Evaluation Infrastructure

We evaluated the applicability and feasibility of our extensions to effect systems by identifying the static and dynamic prevalence of quiescent fields in Java programs selected from the DaCapo benchmark suite [2] and by characterizing the purity of the methods in these programs. The characteristics of the benchmarks we used as inputs for our analyses are given in Figure 1; note that these counts include reachable library classes.

We analyzed these programs with version 0.92 of the GNU Classpath library, using the Soot compiler framework [3] and the DIMPLE$^+$ version of our DIMPLE static analysis tool [4], running on version 5.1.3 of the Yap Prolog system [5]. We

Program	Statements	Classes	Fields	Methods
antlr	1390456	3729	14082	32709
bloat	1413919	3827	14524	33609
eclipse	1384719	3895	15161	33408
hsqldb	1593586	4190	17566	38504
jython	1452433	4058	14737	35604
luindex	1350107	3903	14511	32759
pmd	1509108	4265	15489	36393

Fig. 1. Characteristics of the benchmark programs and transitively reachable library code, including total statement counts, number of analyzed classes, and numbers of declared fields and methods

timed our analyses by running them on one core of a 2 GHZ Opteron workstation with 16 GB of RAM. Finally, we evaluated the dynamic prevalence of quiescing fields by instrumenting the Jikes RVM to record effects on instance fields.

3 Effects and Objects

Type-and-effect systems [6] extend classical type systems so that they characterize not only the values that expressions may assume but also the *computational effects* (reads or writes to shared state) exhibited by evaluating expressions or executing statements and the (abstract) *regions* of the store in which these effects might occur. Lucassen and Gifford's original work on effects systems focused on finding expressions with noninterfering effects in ML-family languages for parallel scheduling, but effects systems have since found a wide range of applications, some of which we review in Section 7.

3.1 Background

Greenhouse and Boyland [7] developed an effects system for object-oriented languages like Java. Their effects system describes READ and WRITE effects that may occur in a hierarchy of regions:

1. The global region *All* contains all mutable state for an entire program,
2. *All* contains *static regions* that model the state of static fields and *instance regions* that contain part or all of the state of individual objects (an object may have several instance regions), and
3. Individual instance regions contain regions corresponding to the state of individual instance fields.

The state of an object may contain the entire state of another object as part of its internal representation. For example, a dictionary object may contain a search tree object that is only accessible from the instance methods of the dictionary object. To address this possibility, Greenhouse and Boyland also provide an

unshared annotation on reference-valued fields. This annotation indicates that any object referred to by an unshared field may only be referred to by that field and thus may be considered logically part of the state of its containing object.

Greenhouse and Boyland present an intraprocedural algorithm to check user-provided effects signatures of methods and to check user-provided *unshared* annotations on object fields, but they do not present an algorithm for reconstructing effect, region, and sharing information for unannotated programs.

3.2 A Lightweight Object-Oriented Effects System

We now present a straightforward effects system and inference algorithm for Java programs. This system is based on that of Greenhouse and Boyland and is deliberately simple in order to clarify subsequent presentation of our novel techniques. While this system is not intended to be particularly sophisticated, our contributions are easily adaptable to more expressive effects systems.

We assume that the Java bytecodes of an input program and its libraries have been preprocessed to generate a conservative approximation of the call graph, a conservative may-alias relation, and the intermediate representation given in Figure 2. In Figure 2, metavariables beginning with s range over statements; S over sets of statements, l over local variables; τ over types; κ over class names; and ν over field names. (We also follow the convention that metavariables with distinct subscripts are assumed to refer to distinct object-level entities.) Note that, as in Java bytecodes, all field names are qualified with the name of their declaring class. Following Greenhouse and Boyland, we treat array loads and stores as accesses to a special field called `[]`. We treat static field loads and stores as accesses to instance fields of a distinguished local l_ω; since we record the declaring class and field name of all field accesses, this sacrifices no precision.

Relation	Description
$\mathsf{formal}(l, i, m)$	Holds when local l respresents the formal parameter at position i (either this or a natural number) in method m.
$\mathsf{actual}(s, l, i)$	Holds when statement s invokes some method with local l as the actual parameter at position i.
$\mathsf{assign}(l_l, l_r)$	Holds when the assignment $l_l = l_r$ occurs in the program.
$\mathsf{load}(s, l, l_h, \kappa.\nu)$	Holds when a heap load statement s reads the value of the $\kappa.\nu$ field from the object referred to by l_h and copies it to l.
$\mathsf{store}(s, l_h, \kappa.\nu, l)$	Holds when a heap store statement s replaces the value of the $\kappa.\nu$ field in the object referred to by l_h with the value of l.
$\mathsf{pt}(l, O)$	Holds when O is the set of abstract objects possibly aliased by l.
$s \in m$	Holds when statement s is part of method body m.
$s \rightarrow m$	Holds when statement s contains a call that may select m, that is, if there is an edge from s to m in the call graph.

Fig. 2. Intermediate representation for simple object-oriented effects inference

The effects annotation on some statement, $\varphi(s)$, consists of READ and WRITE sets of abstract locations. Abstract locations denote sets of concrete locations in which an effect may occur and consist of a pair $\langle \rho, \kappa.\nu \rangle$, where ρ is an abstract region in which an effect may occur and $\kappa.\nu$ describes a field reference qualified by the declaring class of the field. (Because Java is a typed language, a given heap location may be referred to by exactly one kind of field reference.)

Abstract regions consist of (possibly-empty) sets of abstract object identifiers (as given by the may-alias relation pt), the distinguished abstract region $\top$, which includes all possible abstract object identifiers, or special region variables ρ_{this} or $\rho_{0\cdots n}$ denoting the regions reachable from formal parameters; these variables are used to expand method summaries at call sites. In this simple system, we summarize the effects of methods on objects referred to by their parameters but lose precision for objects reachable from the fields of method parameters.

Effect annotations, as pairs of sets, form a semilattice. The join of two effect annotations consists of the READ set of abstract locations formed by unifying the READ sets from each annotation and the WRITE set formed by unifying the WRITE sets from each annotation. Unifying two sets of abstract locations $\mathcal{A}_1$ and $\mathcal{A}_2$, as in a READ or WRITE set, proceeds as follows.

Divide each set $\mathcal{A}_i$ into the two disjoint sets $\mathcal{V}_i$ and $\mathcal{C}_i$ so that $\mathcal{V}_i$ is the set of all abstract locations from $\mathcal{A}_i$ whose regions are region variables, so that $\mathcal{C}_i$ is the set of all abstract locations from $\mathcal{A}_i$ whose regions are sets of abstract object identifiers or $\top$, and so that $\mathcal{V}_i \cup \mathcal{C}_i = \mathcal{A}_i$. $\mathcal{V}_1 \sqcup \mathcal{V}_2$ is defined simply as the union of the two sets. $\mathcal{C}_1 \sqcup \mathcal{C}_2$ consists of the union of the following:

1. The set of locations whose field identifiers appear in $\mathcal{C}_1$ or $\mathcal{C}_2$, but not both:

$$\{\langle \rho, \kappa.\nu \rangle : (\langle \rho, \kappa.\nu \rangle \in \mathcal{C}_1 \wedge \neg\exists \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_2) \vee (\langle \rho, \kappa.\nu \rangle \in \mathcal{C}_2 \wedge \neg\exists \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_1)\}$$

2. The set of locations formed by unifying the regions of each abstract location whose field identifier appears in $\mathcal{C}_1$ and $\mathcal{C}_2$:

$$\{\langle \rho \cup \rho', \kappa.\nu \rangle : \langle \rho, \kappa.\nu \rangle \in \mathcal{C}_1 \wedge \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_2\}$$

We can then define $\mathcal{A}_1 \sqcup \mathcal{A}_2$ as $\mathcal{V}_1 \cup \mathcal{V}_2 \cup (\mathcal{C}_1 \sqcup \mathcal{C}_2)$.

We present the effects inference rules in Figure 3. The relation rpt relates a local variable to its associated region: a region variable for formal parameters, the global region $\top$ for globals, and the set of abstract objects aliased by the local in other cases. The rules READ and WRITE, which establish lower bounds on the effect annotations for load and store statements, are straightforward. SUMMARY gives the annotation summary for a method body; it is this summary that is instantiated at call sites.

The function pmap transforms effects annotations by substituting regions (or region variables) for region variables in a method summary at the point of a call to that method. pmap replaces every region variable with the region variable or explicit region associated with the local of the corresponding actual parameter, as given by the rpt relation.

$$\textsc{r-formal}\ \frac{\mathsf{formal}(l,i,m)}{\mathsf{rpt}(l,\rho_i)} \qquad \textsc{r-global}\ \frac{}{\mathsf{rpt}(l_\omega,\top)} \qquad \textsc{r-other}\ \frac{l \neq l_\omega \qquad \neg\mathsf{formal}(l,i,m) \qquad \mathsf{pt}(l,\rho)}{\mathsf{rpt}(l,\rho)}$$

$$\textsc{read}\ \frac{\mathsf{load}(s,l,l_h,\kappa.\nu) \qquad \mathsf{rpt}(l_h,\rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu\rangle\}\rangle} \qquad \textsc{write}\ \frac{\mathsf{store}(s,l_h,\kappa.\nu,l) \qquad \mathsf{rpt}(l_h,\rho)}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.\nu\rangle\}\rangle}$$

$$\textsc{summary}\ \frac{s_0,\cdots,s_n \in m}{\varphi(m) \sqsupseteq \varphi(s_0) \sqcup \cdots \sqcup \varphi(s_n)} \qquad \textsc{call}\ \frac{s \to m'}{\varphi(s) \sqsupseteq \mathsf{pmap}(s, \varphi(m'))}$$

Fig. 3. Lightweight effects inference rules

4 Initializers and Initialization Effects

Type-and-effect systems identify READ and WRITE effects that code may exhibit upon shared state. (Some, but not all, type-and-effect systems also identify additional effects, such as allocation, exception raising, or taking references.) If the goal of effect systems is to identify potentially interfering computational effects, this taxonomy is rather impoverished: it does not identify *initializations*, which are a special kind of WRITE that will not interfere with any other effects.

4.1 Background and Definitions

We will introduce the notion of initializations with a simple example, but first we provide some background on some properties of Java programs and objects.

Java objects are created via the `new` operator, which performs three tasks before returning a reference to the newly-allocated object: memory allocation, zero-filling object fields, and constructor method invocation. Constructors may invoke other constructors declared in the same class (via the `this()` syntax) or in superclasses (implicitly or explicitly via the `super()` syntax), but there is no way to invoke a constructor on an object after the dynamic lifetime of its constructor invocation completes. There is also no way to create and use an object without invoking its constructor. (This is the case in Java source because `new`, which is the only way to create an object, includes both object allocation and constructor invocation. These tasks correspond to distinct Java bytecode instructions – `new` and `invokespecial` – but the Java Virtual Machine will signal an error if code attempts to access an object that has been allocated but not constructed.) As a consequence, each object will be constructed exactly once before it is accessible to the code that created it.

Consider the `String` class in the Java standard library. `String` is an *immutable* class; once an instance of `String` has been created, its contents cannot be modified. A constructor for the `String` class, in setting up the state of an individual instance, will exhibit WRITE effects on that object's fields. However, these WRITE effects will never interfere with other effects, since the only WRITE

effects on a `String` will occur during its constructor and the code that creates a `String` will not be able to read its state until after the constructor completes.

Immutable classes present an extreme example, but WRITE effects on an object — even a mutable one — by its constructor will not interfere with other effects on that object that occur after the constructor completes. Classical type-and-effect systems do not discriminate between writes that occur to an object during its constructor and writes that occur after an object creation has completed. Such a system may spuriously identify WRITE effects occurring on an object during its creation as interfering with WRITE effects occurring on that object (or on other objects) that have already been created.

We will present a way to discriminate between WRITE effects to objects that have been created and initializations, which are writes that occur to an object while it is being constructed. However, we will first introduce the notion of an *initializer method* and present a algorithm for identifying which methods are initializers for given objects.

4.2 Initializer Methods

Informally, an *initializer method* (or simply an *initializer*) on some object o is a method that executes on o during the dynamic lifetime of its constructor. Since we would like to use the notion of initializer methods to identify WRITE effects that are guaranteed to occur on an object while it is being constructed, we are not interested in any method that merely may initialize part of an object's state; rather, we are interested in methods that may *only* execute on an object during the dynamic lifetime of its constructor.

If we can assume a closed world, we can identify such methods with a simple extension to a conservative static approximation of the program's call graph. We define the *receiver-sensitive call graph* (RSCG) as a set M of nodes corresponding to method bodies, a distinguished *start node* $m_{\mathsf{main}} \in M$, and a set C of labeled call-site edges. An edge is of the form $m \rightarrow_\rho m'$, indicating that m contains a call site that may transfer control to the beginning of m' with a receiver of ρ, which is either this, indicating that m' is an instance method that is invoked on the same object as m, or $\top$, indicating that m' may be invoked on some other method or is not an instance method.

We can thus define a conservative overapproximation of the initializer methods in a program inductively as follows:

1. m is an initializer on o if m is a constructor that may be executed on o (that is, a constructor declared in the class of o or in one of its superclasses).
2. m is an initializer on o if every edge to m in the RSCG is this-labeled and originates from an initializer on o.

In the remainder of this paper we will assume a closed world. We note, however, that this technique is still applicable in an open-world situation — that is, in which the entire program and libraries are not available to be analyzed. It is still possible to identify initializers in an open world as long as the RSCG is constructed in such a way as to include conservative, sound assumptions about

open parts of the program. For example, `private` methods could still soundly be identified as initializers even in an open world, since they can only be invoked from within their declaring class.

4.3 Initialization Effects

An *initialization effect* is a write to an object's state that occurs during the dynamic lifetime of its constructor. Since we have already defined an initializer on some object o as a method that is only transitively invoked through zero or more this-edges in the RSCG from a constructor on o, we can identify initialization effects rather straightforwardly: An initialization effect is a WRITE effect that occurs from within an initializer and on some field of its receiver. We denote sets of initialization effects as an INIT set in an effects annotation and present updated inference rules for initializer methods and for WRITE and INIT effects in Figure 4. (The WRITE rule from Figure 4 supercedes that from Figure 3.)

If we can assume that an object will not be used until after the dynamic lifetime of its constructor, then we can guarantee that a method exhibiting INIT effects in some region ρ will not interfere with methods exhibiting other effects in ρ. This assumption, that uses of an object o by code outside of its constructor will come strictly after the dynamic lifetime of its constructor, is sound only if a reference to o cannot leak to code (say, in another thread) that could effect o before it is fully created. Such leaks are rare (and unidiomatic), so the unsound assumption is perhaps justifiable as a practical matter. For the sake of completeness, though, we briefly sketch a sound treatment of self-leaks:

A value-flow analysis could be used to indicate those constructors that might leak a reference to the constructed object. (In fact, some effect systems track reference leaking explicitly.) The classes containing such constructors could then be considered to not have initializers; as a consequence, WRITE effects occurring during the dynamic lifetime of a constructor on an object of such a class would be conservatively (and soundly) regarded as potentially interfering with `write` effects that occur strictly after the completion of an object's constructor.

Initialization effects are a useful addition to the expressivity of object-oriented effects systems. Since the initializations of a field during an object's creation will not interfere with any reads conducted after the dynamic lifetime of the object's constructor, initialization effects allow effect systems to statically identify

$$\frac{m \text{ is a constructor}}{\mathsf{imeth}(m)}\ \text{IMETH-IMMED} \qquad \frac{(\forall m', \rho) m' \rightarrow_\rho m \models \mathsf{imeth}(m') \wedge \rho = \rho_{\mathsf{this}}}{\mathsf{imeth}(m)}\ \text{IMETH-TRANS}$$

$$\frac{\mathsf{store}(s, l_h, \kappa.\nu, l) \quad \mathsf{rpt}(l_h, \rho) \quad s \in m \quad \neg(\rho = \rho_{\mathsf{this}} \wedge \mathsf{imeth}(m))}{\varphi(s) \sqsupseteq \text{WRITE} : \{\langle \rho, \kappa.\nu \rangle\}\rangle}\ \text{WRITE}$$

$$\frac{\mathsf{store}(s, l_h, \kappa.\nu, l) \quad s \in m \quad \mathsf{imeth}(m) \quad \mathsf{rpt}(l_h, \rho_{\mathsf{this}})}{\varphi(s) \sqsupseteq \text{INIT} : \{\langle \rho_{\mathsf{this}}, \kappa.\nu \rangle\}\rangle}\ \text{INIT}$$

Fig. 4. Inference rules for initialization methods and initializer effects

a greater range of static effects as noninterfering. As we shall see, inferring initialization effects also enables us to identify *quiescing fields*.

5 Quiescing Fields

Some storage is mutable for its entire lifetime, but the lifetimes of many locations can be divided into two phases: an initialization phase, in which the contents of a location are mutable, and a read-only phase, in which the contents of a location will not change. We call such fields *quiescing fields* when the phase transition happens at a statically identifiable and semantically useful place. In this section, we introduce the concept of quiescing fields, explain how we can identify them, and describe why they are useful; compare quiescing fields to Java's `final` fields; and identify the static and dynamic prevalence of quiescing fields in the Java programs from the DaCapo benchmark suite.

5.1 Quiescing Fields Defined and Identified

We define a *quiescing field* as an instance field (i.e. an object member) that is mutable while the object it contains is being constructed but that is immutable for the entire period of program execution strictly after the dynamic lifetime of its containing object's constructor. As a consequence, a quiescing field will have the same value for the entire period that the object containing the field is accessible to the code that created it (and to the rest of the program, modulo the no-leaks assumption of the previous section).

Because a quiescing field is guaranteed not to change after the object that contains it is fully constructed, quiescing fields represent a useful kind of runtime constant. If quiescing fields are prevalent in a program, identifying them can greatly simplify analyses and transformations that require accurate interprocedural data dependence information.

Given sound effects annotations including initialization effects, it is quite straightforward to identify quiescing fields: $\kappa.\nu$ is quiescing if and only if no effect annotation in the whole program contains an abstract location implicating $\kappa.\nu$ (e.g. $\langle \rho, \kappa.\nu \rangle$) in its WRITE set. (If $\kappa.\nu$ is not implicated in the INIT or WRITE sets of any effects annotation, then it is never written after allocation and is trivially a quiescing field.)

Because we need only examine every effect in the whole program once in order to determine which fields are implicated in WRITE effects — and we need not even unify method summaries at call sites in order to do so — quiescing field inference scales linearly with the number of statements in the program.

5.2 Final Fields and Quiescing Fields

The Java language [8] provides the `final` keyword and the semantic guarantee that instance variables declared as `final` will be assigned to exactly once for any given containing object. The `final` keyword thus provides both documentation for programmers and a constraint for use by analyses and transformations.

However, because the guarantee of finality is enforced by a rather coarse flow analysis (identifying "definite assignment," that is, that each final field is on the left-hand side of exactly one assignment along every possible path through each constructor of the object containing it), `final` is of limited applicability. To give one example, since all assignments to `final` fields must occur in the body of a constructor, it is impossible to share initialization code common to several constructors in a `private` instance method.

While it is often possible to restructure the code in a class so that a quiescing field meets the criteria for `final`, such a rewrite may be inconvenient. Furthermore, rewriting code so that a quiescing field is `final` may well obscure the clear meaning of the program for a human reader. Since many programmers will not immediately realize the benefits of having as many fields as possible declared `final`, manual code transformations to expose more fields as `final` are likely to be regarded as insufficiently profitable.

On the contrary, quiescing fields may be written arbitrarily many times during the dynamic lifetime of an object's constructor, not strictly in the static body of the constructor and exactly once along each path of each constructor. Quiescing fields may be read and written freely during the dynamic lifetime of their containing object's constructor, so long as they are not written to after their containing object is fully constructed. Finally, no programmer annotations are necessary to identify quiescing fields, since we present a straightforward and efficient technique for automatically inferring quiescing fields.

5.3 Static and Dynamic Prevalence of Quiescing Fields

We evaluated our definition of quiescing fields on seven of the programs from the DaCapo benchmark suite [2].

We identified the *static prevalence* of `final` and quiescing fields by determining what percentage of all fields implicated in any effect were declared `final`

		Static		*Dynamic*	
Input	Time	% FF	% QF	% FF	% QF
antlr	3.11	19.89	49.25	3.65	24.13
bloat	3.16	22.30	53.01	64.05	70.05
eclipse	3.23	21.50	51.56	77.69	78.53
hsqldb	3.73	18.67	47.97	20.12	58.75
jython	3.61	18.74	52.99	19.17	50.30
luindex	3.06	20.82	51.06	43.87	47.43
pmd	3.35	19.48	48.47	0.78	24.93

Fig. 5. Static and dynamic prevalence of final and quiescing fields in select DaCapo benchmarks. Time represents analysis time in seconds; *static* numbers show the percentage of fields implicated in at least one static effect that are final (FF) and quiescing (QF); *dynamic* numbers indicate the percentage of dynamic reads in a benchmark execution that are of final (FF) and quiescing (QF) fields.

and what percentage were inferred to be quiescing. (Since `final` fields are, by definition, quiescing, counts of quiescing fields include counts of `final` fields.) We also instrumented the Jikes RVM in order to get a trace of all instance field reads from a benchmark execution. From this trace, we derived the percentages of dynamic instance field reads that access `final` and quiescing fields; again, the count of quiescing field reads includes `final` field reads.

Figure 5 gives our complete results; in summary, we found that between 18.7% and 22.3% of fields implicated in any static effects annotation were declared `final`; between 48% and 53% of fields implicated in any static effects annotation were identifiable as quiescing. Between 0.78% and 77.7% of dynamic reads were from `final` fields, and between 24.13% and 78.53% of fields were from quiescing fields. The authors of the bloat, eclipse, and luindex benchmarks seem to have declared a high percentage of frequently-read quiescing fields as `final`; in the other benchmarks, the disparity between the number of dynamic reads of `final` and quiescing fields is much greater.

6 Degrees of Purity

Methods may be *pure*. The classic definition identifies a method that exhibits no effects on mutable state as pure. However, this definition fails to admit idempotent methods that create and modify objects in order to complete their work.

A less restrictive definition, due to Leavens et al. [9,10] and applied for static analysis by Sălcianu and Rinard [11], characterizes a method as pure if and only if it does not modify any state that exists immediately before method entry. This definition of purity captures a notion of method purity as the absence of potential interference with other code: a method may have effects on mutable state that does not exist before it executes. Other definitions of purity are also possible; the concepts we present in this section are generally orthogonal to a base notion of purity and can be straightforwardly adapted to different definitions.

In accepting a definition of purity, we also decide which effects constitute "impure" behavior. Perhaps all side effects are "impure," as in the classical definition. Alternatively, following Leavens et al., we could ignore certain READ or WRITE effects on objects that did not exist at a method's entry. We can then identify some methods as *read-only* – these are methods that may have "impure" READ effects (but not "impure" WRITE effects) on mutable state. (Note that all pure methods are also read-only methods.)

If we are to characterize the purity of methods in typical object-oriented programs, we may wish to characterize instance methods by the effects that they have on mutable state that exists outside of the receiver object.

An *externally-pure* method is one whose "impure" READ or WRITE effects on mutable state occur only to the receiver object (that is, in the ρ_{this} region). Put another way, an externally-pure method is pure, for some definition of "pure," with respect to all state outside of the instance it is operating upon. All pure methods are also externally-pure.

Input	Time	*Externally* % Pure	% RO
antlr	4.47	79.19	81.16
bloat	4.63	77.05	78.40
eclipse	4.63	79.21	80.73
hsqldb	5.44	76.87	78.26
jython	5.25	77.02	78.31
luindex	4.39	80.30	81.89
pmd	4.88	79.05	80.50

Fig. 6. Percentage of all instance methods that are externally- pure or read-only

An *externally-read-only* method is one whose "impure" WRITE effects on durable state occur only to the receiver object or to state that did not exist immediately before method entry. Such a method is read-only with respect to all state outside of the instance it is operating upon. All externally-pure methods are also externally-read-only.

We can combine these notions of purity with initialization effects and quiescing fields by masking INIT effects (which represent writes to the state of newly-allocated objects) and masking READ effects on quiescing fields. If we do so, we can identify a vast preponderance of instance methods as externally-pure or externally-read-only, as in Figure 6.

7 Related Work

Work related to our contributions in this paper falls into two broad categories: work on effects systems and work on inferring fields or memory locations that are immutable for at least some part of their lifetime.

7.1 Effects Systems

Lucassen and Gifford's foundational paper on polymorphic effect systems [6] focused on identifying scheduling constraints for execution of implicitly-parallel programs, but later work has established applications of effects systems in region-based memory management [12], and in automatically providing annotations for a model checker or specification language [11]. We note that our contributions, by improving the precision of effects analyses, can also improve the precision of analyses and transformations that depend on effects annotations.

The natural compatibility of effects and objects has led to a great deal of excellent work; as we discussed in Section 3, Greenhouse and Boyland [7] devised an idiomatic, object-oriented treatment of regions and effects, but did not provide an inference algorithm. Bierman and Parkinson [13] extended the work of Greenhouse and Boyland with a semantic treatment of effects and an effects inference algorithm for a subset of Java; their work left region annotations as

a responsibility for the programmer. (Effect and region reconstruction for functional languages [14] is a better-studied problem.) Most recently, Cherem and Rugina [15] presented a parameterized framework for compact effect signatures, which allows clients of effect annotations to trade precision for annotation size.

Given a notion of effects, it is possible to talk about the purity of functions. Barnett et al. [16] present several definitions of purity in the context of object-language methods that may appear in checkable specifications: observational purity (which admits memoization), strong purity (the classic definition), and weak purity (in which methods may modify newly-allocated state). Sălcianu and Rinard [11] present an analysis to identify weakly-pure methods. Barnett et al. [17] extend the Sălcianu-Rinard analysis to support iterators and the additional features, such as pass-by-reference, of the .NET runtime.

Because it masks INIT effects, our analysis can be used to identify the subset of weakly-pure methods that only exhibit INIT effects on newly-allocated objects; other analyses [11,17] can identify a broader range of weakly pure methods. Consider, for example, a method that constructs a `StringBuffer` and then invokes its `append` method several times before returning a `String` constructed from the buffer; this is weakly pure, but would not be identified as such by our analysis because it exhibits WRITE effects as well as INIT effects. In addition, our purity analysis does not specifically treat iterators.

In general, the work described in this paper is intended to enhance the expressivity and precision of effect systems and purity analyses. It would certainly be possible to extend the system we present with more expressive features, like the information flow effects of Cherem and Rugina or the weak purity of Sălcianu and Rinard. (In particular, treating iterators demands care and a more expressive system.) In addition, one could use the results of a field uniqueness analysis (like that of Ma and Foster [18]) or object inlining transformation in order to automatically generate *unshared* annotations on object fields. Conversely, we believe that initialization effects, quiescing fields, and external purity can be introduced to an extant effect system as crosscutting concerns.

7.2 Inferring Eventual Immutablity

Several analyses (notably Porat et al. [19]) identify Java fields that are immutable, even if the fields are not declared as `final`. Most directly related to our concept of quiescing fields, however, is the *stationary fields* analysis of Unkel and Lam [20], which we shall focus on in the remainder of this review. Unkel and Lam use a flow- and context-sensitive pointer analysis to identify fields for whom every dynamic read must come after every dynamic write.

While both stationary fields and quiescing fields are capable of identifying eventually-immutable fields that are not declared `final`, and both identify about half of all fields in some set of realistic Java programs as eventually-immutable, there are several interesting differences between our approaches. Our approach is substantially more lightweight: we use a flow- and context- insensitive analysis that completes in seconds; their approach uses a flow- and context-*sensitive* analysis that takes between 7 and 106 minutes to analyze a realistic Java program.

However, their approach identifies stationary fields and can also track their referents with greater precision; we merely identify quiescing fields. Therefore, both analyses can be used to improve the precision of side-effect and related analyses; theirs is better-suited to improving the precision of alias analyses.

We note that the definitions of quiescing and stationary fields are subtly incompatible: while it seems most likely that the intersection of the sets of quiescing and stationary fields for any given program would be large, there are quiescing fields that are not stationary fields (e.g. those that might be read in the constructor before a write), and there are stationary fields that are not quiescing fields (e.g. those that are written after the dynamic lifetime of a constructor, but before any use of an object). We believe that investigating the relationships between these kinds of fields — and between the analyses and transformations enabled by identifying each — represents a fruitful avenue for future work.

8 Conclusion

This paper has presented three major contributions that enhance the expressivity and precision of effects systems, purity analyses, and related analyses and transformations: the concepts of initialization effects, quiescing fields, and external method purity, as well as analyses to infer these automatically. In so doing, we have identified great amounts of mostly-functional behavior in the real-world Java programs from the DaCapo benchmark suite. Most notably, our techniques are novel, lightweight, and readily composable with extant systems and analyses.

Acknowledgments. We are grateful to Nicholas Kidd and the anonymous referees for helpful comments on an earlier draft of this paper.

References

1. Knight, T.: An architecture for mostly functional languages. In: LFP 1986: Proceedings of the 1986 ACM conference on LISP and functional programming, pp. 105–112. ACM, New York (1986)
2. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. ACM Press, New York (2006)
3. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p. 13. IBM Press (1999)
4. Benton, W.C., Fischer, C.N.: Interactive, scalable, declarative program analysis: from prototype to implementation. In: PPDP 2007: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 13–24. ACM, New York (2007)

5. Rocha, R., Silva, F., Costa, V.S.: On Applying Or-Parallelism and Tabling to Logic Programs. Theory and Practice of Logic Programming Systems 5, 161–205 (2005)
6. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 47–57. ACM Press, New York (1988)
7. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 205–229. Springer, Heidelberg (1999)
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley, Boston (2000)
9. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. Technical Report TR98-06, Department of Computer Science, Iowa State University, Ames, Iowa (1998)
10. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes 31, 1–38 (2006)
11. Sălcianu, A., Rinard, M.C.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
12. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation (1997)
13. Bierman, G.M., Parkinson, M.J.: Effects and effect inference for a core Java calculus. In: Bono, V., Bugliesi, M. (eds.) Electronic Notes in Theoretical Computer Science, vol. 82. Elsevier, Amsterdam (2003)
14. Talpin, J., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming 2, 245–271 (1992)
15. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: Krishnamurthi, S., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 172–186. Springer, Heidelberg (2007)
16. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44. In: ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP), pp. 11–19 (2004)
17. Barnett, M., Fändrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. In: IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (2007)
18. Ma, K.K., Foster, J.S.: Inferring aliasing and encapsulation properties for java. In: OOPSLA 2007: Proceedings of the 22nd annual ACM SIGPLAN conference On Object Oriented Programming Systems and Applications, pp. 423–440. ACM Press, New York (2007)
19. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in java. In: CASCON 2000: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research, p. 10. IBM Press (2000)
20. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of Java's final fields. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 183–195. ACM, New York (2008)

The Higher-Order Aggregate Update Problem

Christos Dimoulas and Mitchell Wand

College of Computer and Information Science, Northeastern University,
360 Huntington Avenue, Boston, Massachusetts 02115
{chrdimo,wand}@ccs.neu.edu

Abstract. We present a multi-pass interprocedural analysis and transformation for the functional aggregate update problem. Our solution handles untyped programs, including unrestricted closures and nested arrays. Also, it can handle programs that contain a mix of functional and destructive updates. Correctness of all the analyses and of the transformation itself is proved.

1 Introduction

The update of aggregate data structures like arrays is expensive in a functional language because it involves copying the whole data structure. Naively adding destructive update to a functional language does not solve the problem, because the combination loses the compositional properties of functional languages.

A series of papers [1,4,5,7,8,9,12] have pursued the idea of destructive update transformation: an optimization that transforms functional updates into assignments whenever a flow analysis reveals that the array value being updated is dead following the update.

In this paper we present and prove the correctness of an algorithm for destructive update transformation that allows arrays to be nested in arrays and stored in closures. It allows destructive updates and functional updates to coexist in the source program if desired. Furthermore, it does not rely on any type information from the underlying program. To our knowledge, this is the first algorithm with all these features.

The transformation is based on a multi-pass inter-procedural program analysis. We first perform a control-flow analysis, which is used to construct a reachability analysis. We then perform a liveness analysis. The results of these analyses are combined to obtain a live variable analysis.

In section 2 we present a simple example to demonstrate the problem and show the use of our method. In section 3, we give the syntax and the semantics of our language. In section 4, we describe the architecture of our framework, the intermediate layers of analyses and their properties. In section 5 we define the transformation and sketch its correctness proof. Section 6 reviews previous research results in the field.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 44–58, 2009.

2 Examples

Consider the following program where each expression is marked with a unique label, the $\texttt{NEW}(n, v)$ operator creates a new array with n cells of value v and the $\texttt{UPD}(l, i, v)$ operator functionally updates the i-th cell of the array l with the new value v.

$^0(\,^1\lambda\texttt{x}.^2\texttt{UPD}(^3\texttt{x}, ^4\texttt{1}, ^5\texttt{x})$
$\quad{}^6\,(^7\lambda\texttt{y}.^8\texttt{UPD}(^9\texttt{y}, ^{10}\texttt{1}, ^{11}\texttt{444})$
$\quad\quad{}^{12}\,(^{13}\lambda\texttt{f}.^{14}(^{15}\texttt{f}\ \ ^{16}\texttt{333})$
$\quad\quad\quad{}^{17}\lambda\texttt{z}.^{18}\texttt{NEW}(^{19}\texttt{4}, ^{20}\texttt{z}))))$

By constructing the control-flow graph of the program, we observe that the UPD-expression with label 8 is applied on l_1 which is created by the NEW-expression. At the time of the evaluation of the UPD-expression, l_1 is is not reachable from any closures or arrays in the continuation of the UPD-expression. Also, l_1 is not reachable from the new array l_2 that the UPD-operation will return to the program. Thus l_1 is not live after the execution of the UPD-operator. So, replacing the functional update with a destructive one does not change the semantics of the program.

The UPD-expression with label 2 is applied on variable y which is bound to l_2. The result of the UPD-operation l_3 contains l_2. Thus l_2 is live after the evaluation of the UPD-expression and the functional update cannot be replaced by a destructive one. If the UPD-expression was replaced by a destructive update then the original and the transformed program would not agree on the contents of l_2.

From the above, we can conclude that the original program can be transformed to the following one without changing its semantics:

$^0\,(^1\lambda\texttt{x}.^2\texttt{UPD}(^3\texttt{x}, ^4\texttt{1}, ^5\texttt{x})$
$\quad{}^6\,(^7\lambda\texttt{y}.^8\texttt{UPD!}(^9\texttt{y}, ^{10}\texttt{1}, ^{11}\texttt{444})$
$\quad\quad{}^{12}\,(^{13}\lambda\texttt{f}.^{14}(^{15}\texttt{f}\ \ ^{16}\texttt{333})$
$\quad\quad\quad{}^{17}\lambda\texttt{z}.^{18}\texttt{NEW}(^{19}\texttt{4}, ^{20}\texttt{z}))))$

Each time we replace a functional update with a destructive update we avoid copying the array, making our programs more efficient in terms of time. Our framework aims to provide a method that can be used to detect such plausible replacement points in programs by statically predicting if a program variable is live or not.

3 The Language

Our language is a variant of the call-by-value untyped lambda calculus with operators for array manipulation.

Every expression and value comes with a unique label θ. Values v are either basic values $\texttt{c}$, function closures $(\lambda\texttt{x}.E, \rho)$, or memory locations l of arrays of values. Expressions include conditionals, primitive operators p, and array operators g. See figure 1 for details. Our language is untyped, so it can express recursive procedures. Our analysis would of course work if the language were restricted to a typed subset.

$$
\begin{array}{lcl}
E & ::= & {}^{\theta}T \\
\\
T & ::= & \mathtt{x} \mid \mathtt{c} \mid l \mid \lambda \mathtt{x}.E \mid (E_1\ E_2) \mid g(E_1, \ldots) \mid p(E_1, \ldots) \\
 & \mid & \mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2
\end{array}
$$

where $\mathtt{x} \in \mathit{Var}$, $\theta \in \mathit{Lab}$, $\mathtt{c} \in \mathit{Scalar}$, $p \in \mathit{Prim}$, $g \in \{\mathtt{UPD}, \mathtt{UPD!}, \mathtt{NEW}, \mathtt{REF}\}$.

Fig. 1. Language Syntax

$$
\begin{array}{lcl}
S & ::= & \langle \mathit{halted}, v, \Sigma \rangle \mid \langle \alpha, \rho, G, K, \Sigma \rangle \\
\\
G & ::= & E \mid v \\
\\
R & ::= & {}^{\theta}(E\ {}^{\theta}[\,]) \mid {}^{\theta}({}^{\theta}[\,]\ v) \\
 & \mid & {}^{\theta}g(v_1, \ldots, v_{i-1}, {}^{\theta}[\,], E_{i+1}, \ldots, E_n) \mid {}^{\theta}p(v_1, \ldots, v_{i-1}, {}^{\theta}[\,], E_{i+1}, \ldots, E_n) \\
 & \mid & {}^{\theta}\mathtt{if}\ {}^{\theta}[\,]\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \\
\\
K & ::= & \mathit{halt} \mid \langle \alpha, \rho, R, K \rangle \\
\\
v \in V & ::= & {}^{\theta}\mathtt{c} \mid {}^{\theta}l \mid ({}^{\theta}\lambda \mathtt{x}.E, \rho) \\
\\
A & ::= & {}^{\alpha.\theta}\langle v_1, \ldots, v_n \rangle \\
\\
\mathit{Loc} & ::= & \alpha.\theta
\end{array}
$$

where $\alpha, \theta \in N^*$, $l \in \mathit{Loc}$, $\rho \in \mathit{Var} \rightarrow_{\mathit{fin}} V$, $\Sigma \in \mathit{Loc} \rightarrow_{\mathit{fin}} A$.

Fig. 2. Machine Configurations

We use small-step operational semantics with environments ρ, continuations K, and stores Σ [2]. The configurations of our machine and the continuation frames also include a structured computational address that serves as a time stamp [12]. Time stamp $\alpha.i$ marks the beginning of the evaluation of the i-th subexpression of the expression being evaluated at time stamp α. The body of an n-ary procedure is evaluated at time $\alpha.(n+1)$. When a `NEW` operator is executed, a new location is created using the time stamp. The new location is added in the store domain and points to an array containing the specified values. The new array has a label that is equal to the computational address of the new location. This label has two parts: a dynamic one originating from the time-stamp and the label of the `NEW` expression. The last one is also the label of the newly created location, which is returned as the result of the operator. The `REF` operator is also straightforward. The two update operators `UPD` and `UPD!` are the heart of

$$\langle \alpha, \rho, {}^{\theta} g({}^{\theta_1} T_1, {}^{\theta_2} T_2, \ldots, {}^{\theta_n} T_n), K, \Sigma \rangle$$
$$\rightarrow \langle \alpha.1, \rho, {}^{\theta_1} T_1, \langle \alpha, \rho, {}^{\theta} g({}^{\theta_1}[\,], {}^{\theta_2} T_2, \ldots, {}^{\theta_n} T_n), K \rangle, \Sigma \rangle$$

$$\langle \alpha.i, \rho_i, v_i, \langle \alpha, \rho, {}^{\theta} g(v_1, \ldots, v_{i-1}, {}^{\theta_i}[\,], {}^{\theta_{i+1}} T_{i+1}, {}^{\theta_{i+2}} T_{i+2}, \ldots, {}^{\theta_n} T_n), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha.(i+1), \rho, {}^{\theta_{i+1}} T_{i+1}, \langle \alpha, \rho, {}^{\theta} g(v_1, \ldots, v_{i-1}, v_i, {}^{\theta_{i+1}}[\,], {}^{\theta_{i+2}} T_{i+2}, \ldots, {}^{\theta_n} T_n), K \rangle, \Sigma \rangle$$

$$\langle \alpha.2, \rho_2, v, \langle \alpha, \rho, {}^{\theta}\texttt{NEW}({}^{\theta_n} n, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, {}^{\theta} \alpha.\theta, K, \Sigma[\alpha.\theta \rightarrow {}^{\alpha.\theta} \langle v, \ldots, v \rangle] \rangle$$

$$\langle \alpha.2, \rho_2, {}^{\theta_j} j, \langle \alpha, \rho, {}^{\theta}\texttt{REF}({}^{\theta_l} l, {}^{\theta_j}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, v_j, K, \Sigma \rangle$$
$$\text{where } \Sigma(l) = {}^{\alpha'.\theta_l} \langle v_1, \ldots, v_n \rangle$$

$$\langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\texttt{UPD}({}^{\theta_l} l, {}^{\theta_j} j, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, {}^{\theta} \alpha.\theta, K, \Sigma[\alpha.\theta \rightarrow {}^{\alpha.\theta} \langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle] \rangle$$
$$\text{where } \Sigma(l) = {}^{\alpha'.\theta_l} \langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n \rangle$$

$$\langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\texttt{UPD!}({}^{\theta_l} l, {}^{\theta_j} j, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, {}^{\theta_l} l, K, \Sigma[l \rightarrow {}^{\alpha.\theta_l} \langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle] \rangle$$
$$\text{where } \Sigma(l) = {}^{\alpha'.\theta_l} \langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n \rangle$$

Fig. 3. Machine Reduction Rules

our problem. The UPD primitive is similar to the NEW primitive and creates a new location and a new array. The only difference is that the content of the new array is the updated content of the old array. The UPD! primitive does not create a new location. It updates the contents of the array to which the location points and returns the location and its label unchanged. But the label of the array that the location points to changes: the dynamic part of the label is modified to match the time-stamp of the machine state. This way, we can discriminate between two destructive updates to the same array. Figures 2 and 3 present the details of the behavior of the most interesting operators of our language. The rest follow the standard semantics of left-to-right call-by-value evaluation.

Throughout the rest of this paper, we work within a finite universe of expressions $\mathcal{U}$ closed under subexpressions and containing the initial program E_0. The initial configuration $\langle \epsilon, \emptyset, E_0, \mathit{halt}, \emptyset \rangle$ consists of the initial program in the empty environment and store. *Reachable* configurations are those reachable from this initial configuration. Note that if $E_0 \in \mathcal{U}$, then in any reachable configuration, every expression that appears in the configuration will be drawn from $\mathcal{U}$.

4 The Analysis

Our goal is to identify expressions of the form $\texttt{UPD}(\texttt{x}, E_1, E_2)$ and replace them with $\texttt{UPD!}(\texttt{x}, E_1, E_2)$, if we can prove that this does not affect the semantics of the program. A sufficient correctness condition is that there is no alias to x that

is subsequently accessible from other parts of the program. The existence of higher-order functions and nested arrays in our language implies that closures and arrays can hide aliases of locations from the context. For this purpose we use a reachability analysis, which tracks how variables, closures and locations are connected with each other. To build the reachability analysis, we begin with a flow analysis [10].

4.1 The Control Flow Analysis

0-CFA is an analysis for constructing the flow graph of a program with higher order functions based on standard abstract interpretation techniques. Each expression is assigned a unique label. These labels are used as abstract values $\hat{V}$.

The result of the 0-CFA analysis is a function ϕ from labels of expressions and variables to sets of abstract values, i.e. labels of the possible results and bindings. However, the existence of stores demands the extension of the analysis to predict the possible contents of each location in the store. We accomplish that by developing another prediction function σ that describes the shape of the store.

Definition 1 (Store Shape Analysis). *A store shape analysis is a map* σ*:* $\mathrm{Lab} \rightarrow_{fin} \mathcal{P}(\hat{V})$*, mapping a label for an array to a set of abstract values.*

We say σ *describes a store* Σ*,* $\sigma \models \Sigma$*, iff* $\forall l \in dom(\Sigma).\ (\Sigma(l) = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle \implies \forall 1 \leq i \leq n.\ lab(v_i) \in \sigma(\theta))$.

Definition 2 (Control Flow Analysis). *A control flow analysis is a map* ϕ*:* $(\mathrm{Lab} \cup \mathrm{Var}) \rightarrow_{fin} \mathcal{P}(\hat{V})$*, from labels and variables to a set of abstract values.*

We say ϕ *describes an environment* ρ*,* $\phi \models \rho$*, iff* $\forall \mathtt{x} \in dom(\rho).\ lab(\rho(\mathtt{x})) \in \phi(\mathtt{x}) \wedge (\rho(\mathtt{x}) = ({}^{\theta}\lambda \mathtt{y}.E, \rho') \implies \phi \models \rho')$.

We say $\langle \phi, \sigma\rangle$ *describes an expression* ${}^{\theta}T \in \mathcal{U}$ *,* $\langle \phi, \sigma\rangle \models {}^{\theta}T$*, iff for all* ρ*,* Σ*, if* $\phi \models \rho$*,* $\sigma \models \Sigma$*,* $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$ *is a reachable configuration and* $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$ *, then*

1. $lab(v) \in \phi(\theta)$
2. *if* $v = ({}^{\theta'}\lambda \mathtt{x}.E, \rho'')$*, then* $\phi \models \rho''$
3. $\phi \models \rho'$
4. $\sigma \models \Sigma'$

We say $\langle \phi, \sigma\rangle$ *is sound for* $\mathcal{U}$ *iff* $\forall\, E \in \mathcal{U},\ \langle \phi, \sigma\rangle \models E$.

Note that in the reduction $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$, in the above definition, K is the same in both sides. This reduction represents the evaluation of ${}^{\theta}T$ to the value v.

We can find a sound analysis for a universe of expression $\mathcal{U}$ by solving the set constraints $\mathcal{C}[\mathcal{U}]$ presented in figure 4. These constraints are all Horn clauses, and so are solvable by standard techniques. The most interesting are the constraints that apply to the `UPD`, `UPD!` operators. Given a `UPD` expression ${}^{\theta}\mathtt{UPD}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v)$, if $\theta' \in \phi(\theta_a)$, then the new array contains all the values

$$\frac{{}^{\theta}\lambda \mathtt{x}.E \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(a) \quad \frac{{}^{\theta}l \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(b) \quad \frac{{}^{\theta}\mathtt{c} \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(c)$$

$$\frac{{}^{\theta}\mathtt{x} \in \mathcal{U}}{(\phi(\mathtt{x}) \subseteq \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(d)$$

$$\frac{{}^{\theta}({}^{\theta_1}T_1 \ {}^{\theta_2}T_2) \in \mathcal{U} \qquad {}^{\theta'}\lambda \mathtt{x}.{}^{\theta_\lambda}T_\lambda \in \mathcal{U}}{\begin{array}{l}(\theta' \in \phi(\theta_1) \implies (\phi(\theta_2) \subseteq \phi(\mathtt{x}))) \in \mathcal{C}[\mathcal{U}] \\ (\theta' \in \phi(\theta_1) \implies (\phi(\theta_\lambda) \subseteq \phi(\theta))) \in \mathcal{C}[\mathcal{U}]\end{array}}(e)$$

$$\frac{{}^{\theta}\mathtt{if}\ {}^{\theta_0}T_0\ \mathtt{then}\ {}^{\theta_1}T_1\ \mathtt{else}\ {}^{\theta_2}T_2 \in \mathcal{U}}{((\phi(\theta_1) \cup \phi(\theta_2)) \subseteq \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(f) \quad \frac{{}^{\theta}p({}^{\theta_1}T_1, \ldots, {}^{\theta_n}T_n) \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(g)$$

$$\frac{{}^{\theta}\mathtt{NEW}({}^{\theta_n}T_n, {}^{\theta_v}T_v) \in \mathcal{U}}{((\phi(\theta_v) \subseteq \sigma(\theta) \wedge \theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(h)$$

$$\frac{{}^{\theta}\mathtt{REF}({}^{\theta_a}T_a, {}^{\theta_a}T_a) \in \mathcal{U}}{(\theta' \in \phi(\theta_a) \Rightarrow (\sigma(\theta') \subseteq \phi(\theta))) \in \mathcal{C}[\mathcal{U}]}(i)$$

$$\frac{{}^{\theta}\mathtt{UPD}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v) \in \mathcal{U}}{(\theta' \in \phi(\theta_a) \Rightarrow (\sigma(\theta') \subseteq \sigma(\theta) \wedge \phi(\theta_v) \subseteq \sigma(\theta) \wedge \theta \in \phi(\theta))) \in \mathcal{C}[\mathcal{U}]}(j)$$

$$\frac{{}^{\theta}\mathtt{UPD!}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v) \in \mathcal{U}}{(\theta' \in \phi(\theta_a) \Rightarrow (\phi(\theta_v) \subseteq \sigma(\theta') \wedge \theta' \in \phi(\theta))) \in \mathcal{C}[\mathcal{U}]}(k)$$

Fig. 4. Set Constraints for 0-CFA Analysis $\langle \phi, \sigma \rangle$

that are possibly contained in the old array, $(\sigma(\theta') \subseteq \sigma(\theta))$, plus possible results of the last operand $(\phi(\theta_v) \subseteq \sigma(\theta))$. Since the result of the operation is a new location with static label θ, the label is added to the possible results of the operation, $(\theta \in \phi(\theta))$. On the other hand, given a `UPD!` expression, ${}^{\theta}\mathtt{UPD!}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v)$, no new location is created so the constraints just have to add the possible values of the third operand to the possible values of the existing array , $(\phi(\theta_v) \subseteq \sigma(\theta'))$ and add the static label of the updated location to the possible results of the operation $(\theta' \in \phi(\theta))$.

Theorem 1 (Soundness of $\langle \phi, \sigma \rangle$). *If $\langle \phi, \sigma \rangle$ satisfies the constraints $\mathcal{C}[\mathcal{U}]$ in figure 4 then $\langle \phi, \sigma \rangle$ is sound for $\mathcal{U}$.*

Proof. Following [13], we extend the constraints from constraints on expressions to constraints on configurations S. The most interesting new constraints are presented in figure 5. We write $X \in S$ iff X occurs in S. $lab_{[\,]}(R)$ denotes the label of the hole of R, $lab(R)$ denotes the label of the expression of the frame R, and $lab_{[\,]}(K)$ denotes the label of the hole of the top frame of K. We show that

$$\frac{\rho \in \Sigma \qquad \mathtt{x} \in dom(\rho)}{(\phi(lab(\rho(\mathtt{x}))) \subseteq \phi(\mathtt{x})) \in \mathcal{C}[S]}(l) \qquad \frac{\Sigma \in S \qquad l \in dom(\Sigma) \qquad \Sigma(l) = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle \in \Sigma}{(\forall i \leq n.\phi(lab(v_i)) \subseteq \sigma(\theta)) \in \mathcal{C}[S]}(m)$$

$$\frac{\langle \alpha, \rho, R, K\rangle \in S}{(\phi(lab(R)) \subseteq \phi(lab_{[\,]}(K))) \in \mathcal{C}[S]}(n) \qquad \frac{\langle \alpha, \rho, E, \langle \alpha', \rho', R, K\rangle, \Sigma\rangle \in S}{(\phi(lab(E)) \subseteq \phi(lab_{[\,]}(R))) \in \mathcal{C}[S]}(o)$$

Fig. 5. Extended Set Constraints for Constraints 0-CFA Analysis $\langle \phi, \sigma \rangle$

if $S \rightarrow S'$, the constraints for S imply the constraints for S'. Then we obtain the desired result by induction on the length of reduction $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$. ■

Consider the example from section 2. The only constraints relevant to $\phi(\mathtt{y})$ are: $\mathtt{18} \in \phi(\mathtt{18}) \subseteq \phi(\mathtt{14}) \subseteq \phi(\mathtt{12}) \subseteq \phi(\mathtt{y})$. So the smallest solution for ϕ gives $\phi(\mathtt{y}) = \{\mathtt{18}\}$. Similarly, in the smallest solution $\sigma(\mathtt{18}) = \{\mathtt{20}\}$, $\phi(\mathtt{8}) = \{\mathtt{8}\}$, $\phi(\mathtt{x}) = \{\mathtt{8}\}$ and $\phi(\mathtt{2}) = \{\mathtt{2}\}$.

4.2 The Reachability Analysis

The control flow analysis does not describe how values and expressions are associated through the store. In order to describe this relation we use the notion of reachability.

Definition 3 (Reachable Value). *A value w is reachable from a value v in a store Σ,* $\mathrm{reach}(w, v, \Sigma)$*, iff either:*

1. $v = w$*, or*
2. $v = {}^{\theta}l'$ *and* $\Sigma(l') = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle$ *and* $\exists i$ *s.t.* $\mathrm{reach}(w, v_i, \Sigma)$*, or*
3. $v = ({}^{\theta}\lambda \mathtt{x}.{}^{\theta'}T, \rho)$ *and* $\exists \mathtt{y} \in fv(\lambda \mathtt{x}.{}^{\theta'}T)$ *s.t.* $\mathrm{reach}(w, \rho(\mathtt{y}), \Sigma)$.

Following the same path as before, we build an analysis that returns a function $\mathcal{R}$ that associates an expression with the labels of all values reachable from its value.

Definition 4 (Reachability Analysis). *A reachability analysis is a map $\mathcal{R}$:* $(\mathrm{Lab} \cup \mathrm{Var}) \rightarrow_{fin} \mathcal{P}(\hat{V})$*, mapping each label or variable to a set of abstract values.*

We say $\mathcal{R}$ describes a store Σ, $\mathcal{R} \models \Sigma$, iff $\forall l \in dom(\Sigma).(\Sigma(l) = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle \implies \forall 1 \leq i \leq n.(\mathrm{reach}(w, v_i, \Sigma) \implies lab(w) \in \mathcal{R}(\theta)))$.

We say $\mathcal{R}$ describes an environment ρ under a store Σ, $\mathcal{R} \models^{\Sigma} \rho$, iff $\forall \mathtt{x} \in dom(\rho).\ (\mathrm{reach}(w, \rho(\mathtt{x}), \Sigma) \implies lab(w) \in \mathcal{R}(\mathtt{x}))$.

We say $\mathcal{R}$ describes an expression ${}^{\theta}T \in \mathcal{U}$, $\mathcal{R} \models {}^{\theta}T$, iff for all ρ, Σ, if $\mathcal{R} \models^{\Sigma} \rho$, $\mathcal{R} \models \Sigma$, $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$ is a reachable configuration and $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$, then

1. $\text{reach}(l, v, \Sigma')$ *implies* $lab(l) \in \mathcal{R}(\theta)$
2. *if* $v = ({}^{\theta'}\lambda \mathtt{x}.E, \rho'')$, *then* $\mathcal{R} \models^{\Sigma'} \rho''$
3. $\mathcal{R} \models^{\Sigma'} \rho'$
4. $\mathcal{R} \models \Sigma'$

We say $\mathcal{R}$ *is sound for a universe of expressions* $\mathcal{U}$, *iff* $\forall\, E \in \mathcal{U}, \mathcal{R} \models E$.

In order to build a sound $\mathcal{R}$, we use a sound control flow analysis to compute the possible values that an expression may be evaluated to or a variable may be bound to. Then, we perform a sort of transitive closure operation inside the environments and the store.

Theorem 2 (Soundness of $\mathcal{R}$). *Given a sound control flow analysis* $\langle \phi, \sigma \rangle$ *for* $\mathcal{U}$, *define* $\mathcal{R}$ *to be the smallest function (in the partial function ordering) that satisfies the equations*

- $\mathcal{R}(t) = \phi(t) \cup \mathcal{M}(t) \cup \mathcal{N}(t)$
- $\mathcal{M}(t) = \{\theta' | \theta'' \in \sigma(t) \wedge \theta' \in \mathcal{R}(\theta'')\}$
- $\mathcal{N}(t) = \{\theta' | \theta'' \in \phi(t) \wedge {}^{\theta''}\lambda \mathtt{x}.E \in \mathcal{U} \wedge \mathtt{y} \in fv({}^{\theta''}\lambda \mathtt{x}.E) \wedge \theta' \in \mathcal{R}(y)\}$

where $t \in Lab \cup Var$. *Then* $\mathcal{R}$ *is sound for* $\mathcal{U}$.

Proof. By induction on the definition of $reach(w, v, \Sigma)$ using the soundness of $\langle \phi, \sigma \rangle$. ∎

The most interesting part of the definition of $\mathcal{R}$ is the auxiliary set $\mathcal{N}$ that talks about λ-terms, ${}^{\theta}\lambda \mathtt{x}.E$. Then the possibly reachable locations from this term are the locations hidden in all the possible environments that are used for the evaluation of the term. Thus, the reachable locations from the term are the reachable locations from all its free variables, $\forall \mathtt{y} \in fv({}^{\theta}\lambda \mathtt{x}.E).\mathcal{R}(\mathtt{y}) \subseteq \mathcal{R}(\theta)$.

Consider the example from section 2. The only constraints relevent to $\mathcal{R}(\mathtt{y})$ are: $\mathtt{18} \in \mathcal{R}(\mathtt{18}) \subseteq \mathcal{R}(\mathtt{12}) \subseteq \mathcal{R}(\mathtt{y})$. So the smallest solution for $\mathcal{R}$ gives $\mathcal{R}(\mathtt{y}) = \{\mathtt{18}\}$. Also, the only constraints relevant to $\mathcal{R}(\mathtt{8})$ are: $\{\mathtt{8}\} \subseteq \{\mathtt{8}\} \cup \mathcal{R}(\mathtt{16}) \subseteq \mathcal{R}(\mathtt{8})$. So in the smallest solution $\mathcal{R}(\mathtt{8}) = \{\mathtt{8}\}$. Similarly in the smallest solution $\mathcal{R}(\mathtt{x}) = \{\mathtt{8}\}$ and $\mathcal{R}(\mathtt{2}) = \{\mathtt{2}, \mathtt{8}\}$.

4.3 The Liveness Analysis

In order to define the liveness of a location in a machine configuration, we need first to extend the notion of reachability to continuations K.

A value w is reachable from a continuation K in a store Σ if it is reachable in Σ from a value v or variable $\mathtt{x}$ of the top frame of the continuation stack or if it is reachable in Σ by a lower frame of the continuation stack.

Definition 5 (Reachable Value from a Continuation). *Given a store* Σ, *reachability from a continuation* K, $\text{reach}(w, K, \Sigma)$, *is defined as follows.*

- *No value is reachable from halt in any store* Σ.
- w *is reachable from* $\langle \alpha, \rho, R, K \rangle$ *iff either:*

1. $\exists v$ *that occurs in* R *s.t.* $\text{reach}(w, v, \Sigma)$, *or*
2. $\exists \mathtt{x} \in fv(R)$ *s.t.* $\text{reach}(w, \rho(\mathtt{x}), \Sigma)$, *or*
3. $\text{reach}(w, K, \Sigma)$.

A location is live in a configuration iff it is reachable *after* the evaluation of the current expression. If the expression is a value, this means simply that it is reachable from the value itself or from the current continuation.

Definition 6 (Live Location in a Configuration). *Let* $l = \alpha_l.\theta_l$ *and* $lab(l) = \theta_l$. *Liveness in a configuration is defined as follows:*

- $\langle halted, v, \Sigma\rangle$, $l \in dom(\Sigma)$ *is live iff* $\text{reach}({}^{\theta_l}l, v, \Sigma)$
- $\langle \alpha, \rho, v, K, \Sigma\rangle$, $l \in dom(\Sigma)$ *is live iff* $\text{reach}({}^{\theta_l}l, v, \Sigma)$, *or* $\text{reach}({}^{\theta_l}l, K, \Sigma)$
- $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$, $l \in dom(\Sigma)$ *is live iff* $(\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle \Rightarrow$ l *live in* $\langle \alpha', \rho', v, K, \Sigma'\rangle$.

Definition 7 (Liveness Analysis). *A liveness analysis* $\mathcal{Z}$ *is a map from expression labels* θ *to sets of labels.* $\mathcal{Z}$ *is sound iff for each label* θ *and for all reachable configurations* $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$, l *live in the configuration implies* $lab(l) \in \mathcal{Z}(\theta)$ *where* $l = \alpha_l.\theta_l$ *and* $lab(l) = \theta_l$.

Given an expression ${}^{\theta}T$, we wish to enumerate the labels of all the locations that could be live following an evaluation of ${}^{\theta}T$. Assume that ${}^{\theta}T$ occurs in a context $E = {}^{\theta'}g({}^{\theta_1}T_1, \ldots, {}^{\theta_{i-1}}T_{i-1}, {}^{\theta}T, {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n)$.

Every evaluation of ${}^{\theta}T$ occurs as part of a sequence of reduction steps

$$\begin{array}{l}\langle \alpha, \rho, E, K, \Sigma\rangle \\ \xrightarrow{*} \langle \alpha.i, \rho, {}^{\theta}T, \langle \alpha, \rho, g(v_1, \ldots, v_{i-1}, [\,], {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n), K\rangle, \Sigma'\rangle \\ \xrightarrow{*} \langle \alpha.i, \rho, v, \langle \alpha, \rho', g(v_1, \ldots, v_{i-1}, [\,], {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n), K\rangle, \Sigma''\rangle\end{array}$$

We need to enumerate the labels of all locations reachable from v or from $K' = \langle \alpha, \rho, g(v_1, \ldots, v_{i-1}, [\,], {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n, K\rangle$. By the definition of reachability, there are exactly four ways in which a location l could be reachable from v or K':

1. l could be reachable from v. This leads to the constraint $\mathcal{R}(\theta) \subseteq \mathcal{Z}(\theta)$.
2. l could be reachable from one of v_1,...,v_{i-1}. Since each of these v_j is the value of ${}^{\theta_j}T_j$, this leads to the constraint $\mathcal{R}(\theta_j) \subseteq \mathcal{Z}(\theta)$ $(1 \le j \le i-1)$.
3. l could be reachable from the value that a free variable is bound to in ${}^{\theta_{i+1}}T_{i+1}, \cdots, {}^{\theta_n}T_n$. This leads to the constraint $\mathtt{x} \in fv({}^{\theta_j}T_j) \implies \mathcal{R}(\mathtt{x}) \subseteq \mathcal{Z}(\theta)$ $(i+1 \le j \le n)$.
4. l could be reachable from K. This leads to the constraint $\mathcal{Z}(\theta') \subseteq \mathcal{Z}(\theta)$.

A similar analysis applies if ${}^{\theta}T$ appears as an argument of a primitive operator p or as the operator or operand of an application or as the test of an `if`-expression. If ${}^{\theta}T$ appears as the body of a λ-expression ${}^{\theta_\lambda}\lambda\mathtt{x}.{}^{\theta}T$ or as a branch of an `if`-expression only cases 1 and 4 apply. If ${}^{\theta}T$ is the expression in the initial configuration, then only case 1 applies. This is summarized in figure 6.

Theorem 3 (Soundness of $\mathcal{Z}$). *Given a control flow analysis* $\langle \phi, \sigma\rangle$ *and a reachability analysis* $\mathcal{R}$, *both sound for* $\mathcal{U}$, *if* $\mathcal{Z}$ *satisfies the constraints in figure 6 then* $\mathcal{Z}$ *is sound for* $\mathcal{U}$.

$$\frac{{}^{\theta}T \text{ is the initial expression}}{(\mathcal{R}(\theta) \subseteq \mathcal{Z}(\theta)) \in \mathcal{C}[\mathcal{U}]}(a)$$

$$\frac{{}^{\theta_\lambda}\lambda \mathtt{x}.{}^{\theta}T \in \mathcal{U} \quad {}^{\theta'}({}^{\theta_1}T_1 \ {}^{\theta_2}T_2) \in \mathcal{U} \quad \theta_\lambda \in \phi(\theta_1)}{\begin{array}{l}(\mathcal{R}(\theta) \subseteq \mathcal{Z}(\theta)) \ \in \mathcal{C}[\mathcal{U}] \\ (\mathcal{Z}(\theta') \subseteq \mathcal{Z}(\theta)) \in \mathcal{C}[\mathcal{U}]\end{array}}(b)$$

$$\frac{\begin{array}{c}{}^{\theta}({}^{\theta_1}T_1 \ {}^{\theta_2}T_2) \text{ or} \\ {}^{\theta}p({}^{\theta_1}T_1, \ldots, {}^{\theta_{i-1}}T_{i-1}, {}^{\theta_i}T_i, {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n) \text{ or} \\ {}^{\theta}g({}^{\theta_1}T_1, \ldots, {}^{\theta_{i-1}}T_{i-1}, {}^{\theta_i}T_i, {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n) \in \mathcal{U}\end{array}}{\begin{array}{ll}\forall i, 1 \leq i \leq n.(\mathcal{R}(\theta_i) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq n, 1 \leq j \leq i-1.(\mathcal{R}(\theta_j) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq n, i+1 \leq j \leq n.\mathtt{x} \in fv({}^{\theta_j}T_j) \implies (\mathcal{R}(\mathtt{x}) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq n.(\mathcal{Z}(\theta) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}]\end{array}}(c)$$

$$\frac{{}^{\theta}\mathtt{if}\ {}^{\theta_1}T_1\ \mathtt{then}\ {}^{\theta_2}T_2\ \mathtt{else}\ {}^{\theta_3}T_3 \in \mathcal{U}}{\begin{array}{ll}\forall i, 1 \leq i \leq 3.(\mathcal{R}(\theta_i) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall j, 2 \leq j \leq 3.\mathtt{x} \in fv({}^{\theta_j}T_j) \implies (\mathcal{R}(\mathtt{x}) \subseteq \mathcal{Z}(\theta_1)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq 3.(\mathcal{Z}(\theta) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}]\end{array}}(d)$$

Fig. 6. Set Constraints for Liveness Analysis $\mathcal{Z}$

$$\frac{\langle \alpha, \rho, R, K\rangle \in S}{(\mathcal{Z}(lab_{[\,]}(K)) \subseteq \mathcal{Z}(lab(R))) \in \mathcal{C}[S]}(e) \qquad \frac{\langle \alpha, \rho, E, \langle \alpha', \rho', R, K\rangle, \Sigma\rangle \in S}{(\mathcal{Z}(lab_{[\,]}(R)) \subseteq \mathcal{Z}(lab(E))) \in \mathcal{C}[S]}(f)$$

Fig. 7. Extended Set Constraints for Liveness Analysis $\mathcal{Z}$

Proof. We apply the same technique used for the proof of the soundness of the control flow analysis. The most interesting extended constraints are presented in figure 7. ∎

The goal of the live variable analysis is to determine which variables of an expression will be bound to live locations.

Definition 8 (Live Variable Analysis). *A live variable analysis $\mathcal{L}$ is a map from expression labels θ to sets of variables. $\mathcal{L}$ is sound iff for all reachable machine configurations $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$, $\rho(\mathtt{x})$ live in the configuration implies* $\mathtt{x} \in \mathcal{L}(\theta)$.

Theorem 4 (Soundness of $\mathcal{L}$). *Given a flow analysis $\langle \phi, \sigma\rangle$ for $\mathcal{U}$ and a liveness analysis $\mathcal{Z}$, both sound for $\mathcal{U}$, $\mathcal{L}(\theta) = \{\mathtt{x} \in fv(\theta) | (\phi(\mathtt{x}) \cap \mathcal{Z}(\theta)) \neq \emptyset\}$ is sound for $\mathcal{U}$.*

Proof. Straightforward by the definitions of live variable analysis and soundness of $\mathcal{Z}$. ∎

Consider the example from section 2. The constraints relevant to $\mathcal{R}(8)$ are: $\mathcal{Z}(6) \cup \mathcal{R}(8) \subseteq \mathcal{Z}(8)$, $\{8\} \subseteq \mathcal{R}(8)$. But in the smallest solution $\mathcal{Z}(6) = \emptyset$, $\mathcal{Z}(8) = \{8\}$ and $\phi(y) = \{18\}$. So $\mathcal{L}(8) = \emptyset$. Similarly, $\mathcal{Z}(0) \cup \mathcal{R}(2) \subseteq \mathcal{Z}(2)$. Again in the smallest solution, $\mathcal{Z}(0) = \emptyset$, $\mathcal{R}(8) = \{2, 8\}$, $\phi(x) = \{8\}$. So $\mathcal{L}(2) = \{x\}$.

5 Replacing Functional with Destructive Updates

5.1 The Transformation

The soundness of the live variable analysis guarantees that if a free variable x occurs in an expression ${}^{\theta}T$ and $\rho(x)$ is live after the evaluation of ${}^{\theta}T$ in some configuration, then $x \in \mathcal{L}(\theta)$. In the case where ${}^{\theta}T = {}^{\theta}\mathtt{UPD}({}^{\theta_x}x, {}^{\theta_1}T_1, {}^{\theta_2}T_2)$, we infer that if $x \notin \mathcal{L}(\theta)$, then x is bound to a location l that is not live after the evaluation of ${}^{\theta}T$. So we can replace the functional update, UPD, with a destructive one, UPD!, without affecting the meaning of the program that ${}^{\theta}T$ appears in, because l is not accessible by any other part of the program.

Definition 9 (The Transformation $(-)^*$). *Let $E \in \mathcal{U}$ and $\mathcal{L}$ a sound live variable analysis for $\mathcal{U}$. Also let Θ be a set of labels s.t. every $\theta \in \Theta$ labels an update of the form ${}^{\theta}\mathtt{UPD}({}^{\theta_x}x, E_1, E_2)$, where $x \notin \mathcal{L}(\theta)$. Then E^* is the result of replacing ${}^{\theta}\mathtt{UPD}({}^{\theta_x}x, {}^{\theta_1}T_1, {}^{\theta_2}T_2)$ by ${}^{\theta}\mathtt{UPD!}({}^{\theta_x}x, {}^{\theta_1}T_1{}^*, {}^{\theta_2}T_2{}^*)$ for each $\theta \in \Theta$.*

In the example from section 2, from the results of the live variable analysis on the program, we concluded that $y \notin \mathcal{L}(8)$. So $\Theta = \{8\}$ and the transformation gives us the expected result from section 2.

5.2 Correctness Proof

We claim that the initial and the transformed program have the same observable behavior.

In order to prove our claim, we define a similarity relation $\sim$ between two configurations. The similarity relation is parameterized by a one-to-one function f that records the correspondence between locations on the left and locations on the right. The relation $\sim^f$ is defined by induction on the various structures involved. The function f avoids the need for a coinductive definition. The key portions of the definition of $\sim$ are shown in figure 8.

Two configurations are similar iff there is a one-to-one function f that makes each of their components similar mod f. Similarity of environments is always done relative to a set of variables Y; ρ and ρ^* are similar mod f iff for each $x \in Y$, their values are similar mod f. Two stores Σ and Σ^* are similar mod f iff for each $(l, l^*) \in f$, $\Sigma(l)$ and $\Sigma^*(l^*)$ are arrays of the same length whose components are similar mod f. Two locations are similar mod f iff they are related by the function f. All the other cases are defined by the obvious structural recursion, except that an UPD-term is similar to an UPD!-term if they have the same label

- $\langle \alpha, \rho, E, K, \Sigma \rangle \sim \langle \alpha^*, \rho^*, E^*, K^*, \Sigma^* \rangle$ iff $\alpha = \alpha^*$ and there exists a one-to-one function $f : (D \subseteq dom(\Sigma)) \to dom(\Sigma^*)$ such that $\rho \sim^f_{fv(E)} \rho^*$, $\Sigma \sim^f \Sigma^*$, $E \sim^f E^*$, and $K \sim^f K^*$
- $\rho \sim^f_Y \rho^*$ iff $\forall \mathtt{x} \in Y.\ \rho(\mathtt{x}) \sim^f \rho^*(\mathtt{x})$
- $\Sigma \sim^f \Sigma^*$ iff $(l, l^*) \in f$ implies $\Sigma(l) = \langle v_1, \ldots, v_n \rangle$, $\Sigma^*(l^*) = \langle v_1^*, \ldots, v_n^* \rangle$ and $\forall i \leq n.\ v_i \sim^f v_i^*$.
- ${}^{\theta}\mathtt{c} \sim^f {}^{\theta^*}\mathtt{c}^*$ iff $\mathtt{c} = \mathtt{c}^*$.
- ${}^{\theta}l \sim^f {}^{\theta^*}l^*$ iff $f(l) = l^*$.
- ${}^{\theta}\mathtt{UPD}(E_l, E_j, E_v) \sim^f {}^{\theta}\mathtt{UPD!}(E_l^*, E_j, E_v^*)$ iff $E_l \sim^f E_l^*$, $E_j \sim^f E_j^*$, $E_v \sim^f E_v^*$ and $\theta \in \Theta$

Fig. 8. The Similarity Relation $\sim$ (selected cases)

$\theta \in \Theta$ and their subterms are similar. Similar expressions always have the same label, unless they are $\sim^f$-related locations.

Clearly, the initial states of the original and the transformed program are similar, using the empty function for f. We then prove, by induction on the length of the computation, that as the original and transformed program compute, they stay in similar configurations. Therefore, the machines halt with similar values: if the values are constants, then they must be the same constant.

There are two non-trivial cases: when the machines are at an (UPD!, UPD!) pair (lemma 2), and when they are at a (UPD, UPD!) pair (lemma 6). The former illustrates why f must be injective, and the latter is a point at which the transformation has been applied.

In the first case, we use the one-to-one property of f to show that the destructive updates do not disturb the similarity relation of the produced configurations.

Consider the two following configurations:

$$S = \langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\mathtt{UPD!}({}^{\theta_l}l, j, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle,$$
$$S^* = \langle \alpha.3, \rho_3^*, v^*, \langle \alpha, \rho^*, {}^{\theta}\mathtt{UPD!}({}^{\theta_{l^*}}l^*, j^*, {}^{\theta_v}[\,]), K^* \rangle, \Sigma^* \rangle.$$

Assume that $S \sim^f S^*$. By the semantics of the language we know that $S \to S'$ and $S^* \to S^{*\prime}$ where

$$S' = \langle \alpha, \rho, {}^{\theta_l}l, K, \Sigma' \rangle,$$
$$S^{*\prime} = \langle \alpha, \rho^*, {}^{\theta_{l^*}}l^*, K^*, \Sigma^{*\prime} \rangle$$
where $\Sigma' = \Sigma[l \to {}^{\alpha.\theta_l}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle]$,
$\Sigma^{*\prime} = \Sigma^*[l^* \to {}^{\alpha.\theta_{l^*}}\langle v_1^*, \ldots, v_{j-1}^*, v^*, v_{j+1}^*, \ldots, v_n^* \rangle]$

Lemma 1. $\Sigma' \sim^f \Sigma^{*\prime}$.

Proof. From $S \sim^f S^*$, we know that ${}^{\theta_l}l \sim^f {}^{\theta_{l^*}}l^*$, $\forall 0 \leq i \leq n.\ v_i \sim^f v_i^*$ and $v \sim^f v^*$. Furthermore, f is an one-to-one function. So there is no other $l' \in dom(\Sigma)$ or $l^{*\prime} \in dom(\Sigma^*)$ such that $l' \sim^f l^*$ or $l \sim^f l^{*\prime}$. Since $\Sigma \sim^f \Sigma^*$ we can conclude that $\Sigma[l \to {}^{\alpha.\theta_l}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle] \sim^f \Sigma^*[l^* \to {}^{\alpha.\theta_{l^*}}\langle v_1^*, \ldots, v_{j-1}^*, v^*, v_{j+1}^*, \ldots, v_n^* \rangle]$. ∎

Lemma 2. $S' \sim^f S^{*\prime}$.

Proof. Straightforward by $S \sim^f S^*$ and lemma 1. ∎

In the second case, we take advantage of the definition of the transformation and the fact that this case arises only when the updated location is not live. We define a new one-to-one function f' which makes the two resulting configurations similar.

Consider the two following configurations:

$S = \langle \alpha, \rho, {}^{\theta}\texttt{UPD}(\texttt{x}, E_j, E_v), K, \Sigma \rangle$,
$S^* = \langle \alpha, \rho^*, {}^{\theta}\texttt{UPD!}(\texttt{x}, E_j^*, E_v^*), K^*, \Sigma^* \rangle$

where $\texttt{x} \notin \mathcal{L}(\theta)$ and $S \sim^f S^*$. Assume that $S \xrightarrow{*} S''$ and $S^* \xrightarrow{*} S^{*\prime\prime}$, where
$S'' = \langle \alpha, \rho, {}^{\theta}\alpha.\theta, K, \Sigma'' \rangle$,
$S^{*\prime\prime} = \langle \alpha, \rho, {}^{\theta_{l^*}} l^*, K^*, \Sigma^{*\prime\prime} \rangle$
and where $\forall 1 \leq i \leq n.\ v_i \in \Sigma'(l)$, $\forall 1 \leq i \leq n.\ v_i^* \in \Sigma^{*\prime}(l^*)$,

$$\Sigma'' = \Sigma'\ [\alpha.\theta \rightarrow {}^{\alpha.\theta}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle],$$
$$\Sigma^{*\prime\prime} = \Sigma^{*\prime}[l^* \rightarrow {}^{\alpha.\theta_{l^*}}\langle v_1^*, \ldots, v_{j-i}^*, v^*, v_{j+1}^*, \ldots, v_n^* \rangle].$$

Then there must be configurations

$S' = \langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\texttt{UPD}({}^{\theta_l} l, {}^{\theta_j} j, {}^{\theta_v}[\,]), K \rangle, \Sigma' \rangle$,
$S^{*\prime} = \langle \alpha.3, \rho_3^*, v^*, \langle \alpha, \rho^*, {}^{\theta}\texttt{UPD!}({}^{\theta_{l^*}} l^*, {}^{\theta_j} j, {}^{\theta_v}[\,]), K^* \rangle, \Sigma^{*\prime} \rangle$

such that $S \xrightarrow{*} S' \rightarrow S''$ and $S^* \xrightarrow{*} S^{*\prime} \rightarrow S^{*\prime\prime}$. Assume that $S' \sim^f S^{*\prime}$.

Consider now the following f':

- $f'(\alpha.\theta) = l^*$
- if l' live in S'' and $l' \neq \alpha.\theta$ then $f'(l') = f(l')$.

Lemma 3. *f' is a one-to-one function.*

Proof. By $S' \sim^f S^{*\prime}$, $f(l) = l^*$. f is a one-to-one function. Let g be the function defined by the second branch of the definition of f'. g is a subset of f. Since l is not live in in S'', g is a one-to-one function that does not include the pair (l, l^*). Also, there is no $l' \in dom(\Sigma')$ such that $l' \neq l$ and $f(l') = l^*$. So there is no $l' \in dom(\Sigma')$ such that $g(l') = l^*$. The extension of g with the pair $(\alpha.\theta, l^*)$ defines f'. From the above we can conclude that f' is a one-to-one function. ∎

Lemma 4. *$\forall w, w^*$ if* $\text{reach}(w, K, \Sigma'')$ *and $w \sim^f w^*$ then $w \sim^{f'} w^*$.*

Lemma 5. *Let $\Sigma''(\alpha.\theta) = {}^{\alpha.\theta}\langle w_1, \ldots, w_n \rangle$.*
$\forall 1 \leq i \leq n, w, w^*$ if $\text{reach}(w, w_i, \Sigma'')$ *and $w \sim^f w^*$ then $w \sim^{f'} w^*$.*

Proof. We prove these two lemmas by induction on the depth of w. The interesting part is in the base case of the inductive proof, when w is a location, where we proceed by case analysis on whether w is the updated location l. ∎

Lemma 6. *$S'' \sim^{f'} S^{*\prime\prime}$.*

Proof. By lemmas 4 and 5, we can conclude that all the values that are reachable from S'' are related through f' with the corresponding values that occur in $S^{*\prime\prime}$.

Thus by lemma 3 and the definition of the similarity relation all the elements of the two resulting configurations are similar. ■

Observe that these lemmas imply that similar configuration either both halt or both take a step to similar configurations. Combining these lemmas gets us:

Theorem 5 (Correctness of $(-)^*$ - The main theorem). *Let E_0 be the initial program, and let E_0^* be the result of applying the transformation on E_0. Then $\langle \epsilon, \emptyset, E_0, halt, \emptyset \rangle \xrightarrow{n} \langle halted, v, \Sigma \rangle$ iff $\langle \epsilon, \emptyset, E_0^*, halt, \emptyset \rangle \xrightarrow{n} \langle halted, v^*, \Sigma^* \rangle$ where for some one-to-one function f $v \sim^f v^*$ and $\Sigma \sim^f \Sigma^*$.*

6 Related Work

Our effort is strongly related to previous work on inter-procedural aggregate update analysis. Hudak and Bloss [4,5] propose an aggregate update analysis for strict first-order languages with flat arrays. Their approach combines abstract interpretation with conventional flow analysis. In their turn, Draghicescu and Purushothaman [1] presented an aggregate optimization for non-strict first-order languages with flat arrays. The transformation is based on a liveness analysis.

The analysis of Wand and Clinger [12], which extends the analysis of [7,8], presents a modular framework. They build a propagation analysis, and on top of that an alias analysis and finally a live variables analysis. The transformation replaces functional updates of dead variables with destructive updates. Their analysis deals only with first-order languages with arrays of scalar values and they prove the soundness of their analysis using environmental semantics and store erasure.

Shankar [9] proposed a method for safe destructive update in strongly-typed higher-order functional languages with eager order of evaluation. The structure of his framework is very similar to that of [12]: it uses an alias analysis to create a live variable analysis and then uses the results of the analysis to perform the transformation. The analysis handles higher-order programs by means of a fixed point calculation of the live variables. However the analysis cannot handle nested arrays.

From a different perspective, efforts like [3,6] are influenced by the generalization of linear type systems [11] and try to solve the problem using type annotations. None of these can handle untyped programs as our does.

Our optimization is based on the analysis of [12]. We share the spirit of a modular framework which consists of layers with different analysis in each layer computed symbolically. But our analysis differs from that of [12] in a number of ways. Instead of a propagation and an alias analysis we use control-flow analysis [10]. This makes our framework capable of handling higher-order languages and arrays of any kind of values. Also, we can handle source code with both functional and destructive updates. Finally we use a different proof technique for the correctness of the transformation by constructing a bisimulation of the initial and the transformed program based on store shape properties.

References

1. Draghicescu, M., Purushothaman, S.: A uniform treatment of order of evaluation and aggregate update. Theoretical Computer Science 118(2), 231–262 (1993)
2. Felleisen, M., Friedman, D.P.: A calculus for assignments in higher-order languages. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, p. 314 (1987)
3. Guzman, J.C., Hudak, P.: Single-threaded polymorphic lambda calculus. In: IEEE Symposium on Logic in Computer Science, pp. 333–343 (1990)
4. Hudak, P., Bloss, A.: The aggregate update problem in functional programming systems. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 300–314 (1985)
5. Hudak, P., Bloss, A.: Avoiding copying in functional and logic programming languages. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 300–314 (1985)
6. Odersky, M.: How to make destructive updates less destructive. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 25–26 (1991)
7. Sastry, A.V.S.: Efficient Array Update Analysis of Strict Functional Languages. PhD thesis, Computer and Information Science, University of Oregon (1994)
8. Sastry, A.V.S., Clinger, W.D., Ariola, Z.: Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In: Conference on Functional Programming Languages and Computer Architecture, pp. 266–275 (1993)
9. Shankar, N.: Static analysis for safe destructive updates in a functional language. In: International Workshop on Logic-based Program Synthesis and Transformation, pp. 1–24 (2002)
10. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU-CS-91-145 (1991)
11. Wadler, P.: Linear types can change the world! In: IFIP TC 2 Working Conference on Programming Concepts and Methods, pp. 347–359 (1990)
12. Wand, M., Clinger, W.D.: Set constraints for destructive array update optimization. Journal of Functional Programming 11(3), 319–346 (2001)
13. Williamson, G.B.: Flow analysis for higher-order multithreaded computations. PhD thesis, College of Computer and Information Science, Northeastern University, Boston, Massachusetts (2004)

An Abort-Aware Model of Transactional Programming

Kousha Etessami[1,⋆] and Patrice Godefroid[2]

[1] University of Edinburgh
kousha@inf.ed.ac.uk
[2] Microsoft Research
pg@microsoft.com

Abstract. There has been a lot of recent research on transaction-based concurrent programming, aimed at offering an easier concurrent programming paradigm that enables programmers to better exploit the parallelism of modern multi-processor machines, such as multi-core microprocessors. We introduce *Transactional State Machines* (TSMs) as an abstract finite-data model of transactional shared-memory concurrent programs. TSMs are a variant of concurrent boolean programs (or concurrent extended recursive state machines) augmented with additional constructs for specifying potentially nested transactions. Namely, some procedures (or code segments) can be marked as *transactions* and are meant to be executed "atomically", and there are also explicit *commit* and *abort* operations for transactions. The TSM model is non-blocking and allows interleaved executions where multiple processes can simultaneously be executing inside transactions. It also allows nested transactions, transactions which may never terminate, and transactions which may be aborted explicitly, or aborted automatically by the run-time environment due to memory conflicts.

We show that concurrent executions of TSMs satisfy a correctness criterion closely related to serializability, which we call *stutter-serializability*, with respect to shared memory. We initiate a study of model checking problems for TSMs. Model checking arbitrary TSMs is easily seen to be undecidable, but we show it is decidable in the following case: when recursion is exclusively used inside transactions in all (but one) of the processes, we show that model checking such TSMs against all stutter-invariant ω-regular properties of shared memory is decidable.

1 Introduction

There has been a lot of recent research on transaction-based concurrent programming, aimed at offering an easier concurrent programming paradigm that enables programmers to better exploit the parallelism of modern multi-processor machines, such as multi-core microprocessors. Roughly speaking, transactions are marked code segments that are to be executed "atomically". The goal of such

⋆ The work of this author was done partly while visiting Microsoft Research.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 59–73, 2009.

research is to use transactions as the main enabling construct for shared-memory concurrent programming, replacing more conventional but low-level constructs such as locks, which have proven to be hard to use and highly error prone. High-level transactional code could in principle then be compiled down to machine code for the shared memory-machine, as long as the machine provides certain needed low-level atomic operations (such as atomic *compare-and-swap*). Already, a number of languages and libraries for transactions have been implemented (see, e.g., [18] which surveys many implementations).

Much of this work however lacks precise formal semantics specifying exactly what correctness guarantees are provided by the transactional framework. Indeed, there often appears to be a tension between providing strong formal correctness guarantees and providing an implementation flexible and efficient enough to be deemed useful, the latter being usually the main concern of the transactional-memory (TM) research community. When formal semantics is discussed, it is usually to offer an abstract characterization of some specific low-level TM implementation details: such semantics are *distinguishing low-level semantics* in the sense that they typically distinguish some newly proposed implementation from all other previous implementations. Even if transactional constructs were themselves given clear semantics, there would remain the important task of verifying specific properties of specific transactional programs.

The aim of this paper is to provide a state-machine based formal model of transactional concurrent programs, and thus to facilitate an abstract framework for reasoning about them. In order for such a model to be useful, firstly, it must be close enough to existing transactional paradigms so that, in principle, such models could be derived from actual transactional programs via a process of abstraction akin to that for ordinary programs. Secondly, the model should be simple enough to enable (automated) reasoning about such programs. Thirdly, the model should be abstract enough to allow verification of properties of transactional programs independently of any specific TM implementation; the model should thus capture a *unifying high-level semantics* formalizing the view of transactional programmers (unlike most distinguishing low-level semantics discussed in the TM research literature, which represent views of TM implementers).

So, what is a "transaction"? Syntactically, transactions are marked code segments, e.g., demarcated by "`atomic` {...}", or, more generally, they are certain procedures which are marked as `transactional`. (Simple examples of transactional concurrent shared-memory programs are given in Figure 1. These examples will be discussed later.) But what is the semantics? The most common unifying high-level semantics is the so-called "*single-lock semantics*" (see, e.g., [18]), which says that during concurrent execution each executed transaction should appear "as if" it is executing serially without any interleaving of the operations of that transaction with other transactions occurring on other processes. In other words, it should appear "as if" executing each transaction requires every process to acquire a single global transaction lock and to release that lock only when the transaction has completed. The problem with this informal semantics

has to do with precisely what is meant by "as if". A semantics which literally assumes that every concurrent execution proceeds via a single lock, rules out any interleaving of transactions on different processes. It also violates the intended non-blocking nature of the transactional paradigm, and ignores other features, such as the fact that transactions may not terminate, and that typically transactions can be *aborted* either explicitly by the program or automatically by the run-time system due to memory conflicts.

Of course, designers of transactional frameworks would object to this literal interpretation of "as if". Rather, a weaker semantics is intended, but phrasing a simple unifying high-level formal semantics which captures precisely what is desired and leaves sufficient flexibility for an efficient implementation is itself a non-trivial task. Standard correctness notions such as *serializability*, which are used in database concurrency control, are not directly applicable to this setting without some modification. This is because in full-fledged concurrent programming it is no longer the case that every operation on memory is done via a transaction consisting of a block of (necessarily terminating) straight-line code. The "transactional program" running on each process may consist of a mix of transactional and non-transactional code, transactions may be nested, and moreover some transactions (which are programs themselves) may never halt. When adapting correctness criteria to this setting, one needs to take careful account of all these subtle differences.

The key role that aborts play in transactional programming should not be underestimated. Consider a transactional program for reserving a seat on a flight. The program starts a transaction, reads shared memory to see if seats are available and if so, attempts to write in shared memory to reserve a specific seat. If the flight is full or if there is a runtime memory conflict to reserve that specific seat, the transaction must be aborted, and the transactional program must be notified of this abort in order to take appropriate recovery actions. In particular, always forcing each abort to trigger a retry is not a viable option in practice (if the flight is full there is no point retrying forever to book a seat on that flight). So there *must* be some abort mechanism, either through explicit aborts or automatic aborts (or both), which is *not* equivalent to a retry. In other words, those aborts *must be visible* to the transactional programmer and therefore they *must be given a semantics*. As another example, consider transactional programs operating under stringent timing constraints. The programmer may not wish to do arbitrarily many retries after an automatic abort, depending on the current program state. We emphasize these points because earlier feedback we have received suggests that some people in the TM community believe it is adequate to provide the transactional programmer with a high-level semantic model (e.g., single-lock semantics) which does not at all expose them to the possibility of aborts. We believe this is an oversimplification that will only lead to greater confusion for programmers.

In this paper, we propose *Transactional State Machines* (TSMs) as an abstract finite-data model of transactional shared-memory concurrent programs. The TSM model is non-blocking and allows interleaved executions where multiple

processes can simultaneously be executing inside transactions. It also allows nested transactions and transactions which may never terminate.

Using TSMs as a formalization vehicle, we propose a new *abort-aware* unifying high-level semantics which extends the traditional single-lock semantics by allowing the modeling of transactions aborted either explicitly in the program or automatically by the underlying TM implementation. Our abort-aware semantics exposes both explicit and automatic aborts, but it can easily be adjusted to treat automatic aborts as retries.

We define *stutter-serializability*, which we feel captures in a clean and simple way a desired correctness criterion, namely *serializability with respect to committed transactions*, which is (trivially) enjoyed by the single-lock semantics (since no transactions ever abort). We show that our abort-aware TSM semantics preserves this property, while also accommodating aborted transactions.

Finally, we also study model checking of TSMs. We show that, although model checking for general TSMs is easily seen to be undecidable, it is decidable for an interesting fragment. Namely, when recursion is exclusively used inside transactions in all (but one) of the processes, we show that model checking such TSMs against all stutter-invariant ω-regular properties of shared memory is decidable. This decidability result also holds for several other variants of the abort-aware TSM semantics.

2 Overview of the Abort-Aware TSM Semantics

Our abort-aware TSM semantics is based on two natural assumptions which are close in spirit to assumptions used in transactional memory systems. First, we implicitly assume the availability of an atomic (hardware or software implemented) multi-word *compare-and-swap* operation, $CAS(\bar{x}, \bar{x'}, \bar{y}, \bar{y'})$, which compares the contents of the vector of memory locations $\bar{x}$ to the contents of the vector of memory locations $\bar{x'}$, and if they are the same, it assigns the contents of the vector of memory locations $\bar{y'}$ to the vector of memory locations $\bar{y}$. How such an atomic CAS operation is implemented is irrelevant to the semantics. (It can, for instance, be implemented in software using lower-level constructs such as locks blocking other processes.) Second, we assume a form of *strong isolation* (*strong atomicity*). Specifically, there must be minimal atomic operation units on all processes, such that these atomic units are indivisible in a concurrent execution, meaning that a concurrent execution must consist precisely of some interleaved execution of these atomic operations from each process. Thus "atomicity" of operations must hold at some level of granularity, however small. Without such an assumption, it is impossible to reason about asynchronous concurrent computation via an interleaving semantics, which is what we wish to do.

Based on these two assumptions, we can now give an informal description of the abort-aware TSM semantics. TSMs are concurrent boolean programs with procedures, except that some procedure calls may be *transactional* (and such calls may also be nested arbitrarily). Transactional calls are treated differently at run time. After a transactional call is made, the first time any part of shared

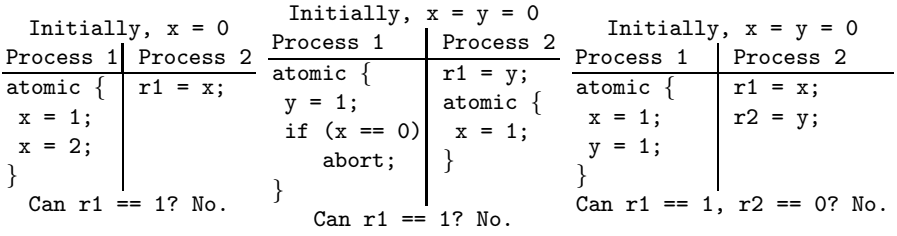


Fig. 1. Examples

memory is used in that transaction, it is copied into a *fixed* local copy on the stack frame for that transaction. A separate, *mutable*, copy (valuation) of shared variables is also kept on the transactional stack frame. All read/write accesses (after the first use) of shared memory inside the transaction are made to the mutable copy on the stack, rather than to the universal copy. Each transaction keeps track (on its stack frame) of those shared memory variables that have been *used* or *written* during the execution of the transaction. Finally, if and when the transaction terminates, we use an atomic *compare-and-swap* operation to check that the current values in (the used part of) the universal copy of shared memory are exactly the same as their fixed copy on the stack frame, and if so, we copy the new values of *written* parts of shared memory from the mutable copy on the stack frame to their universal copy. Otherwise, i.e., if the universal copy of used shared memory is inconsistent with the fixed copy for that transaction, we have detected a memory conflict and we abort that transaction.

The key point is this: if the compare-and-swap operation at the end of a transaction succeeds and the transaction is not aborted, then we can in fact "schedule" the entire activity of that transaction inside the "infinitesimal time slot" during which the atomic compare-and-swap operation was scheduled. In other words, there exists a serial schedule for non-aborted transactions, which does not interleave the operations of distinct non-aborted transactions with each other. This allows us to establish the *stutter-serializability* property for TSMs.

The above description is over-simplified because, e.g., TSMs also allow nested transactions and there are other technicalities, but it does describe some key aspects of the model. We describe the model in a bit more detail in Section 3. Due to space constraints, the full formal model is described in the tech report [12]. We show that TSMs are stutter-serializable in Section 4. We study model checking for TSMs in Section 5, and show that, although model checking for general TSMs is undecidable, there is an interesting fragment for which it remains decidable.

Examples. Figure 1 contains simple example transactional programs (adapted from [13]). Transactions are syntactically defined using the keyword `atomic`. With each example, we describe the possible effect, in our TSM model, on the variables r_1 (and r_2) at the end of the example's execution. As mentioned, in the TSM model the execution of transactions on multiple processes can interleave, and moreover the execution of transactional and non-transactional code can also interleave. So, in the leftmost example, what happens if the non-transactional

code executed by Process 2 executes before the transaction on Process 1 has completed? In the TSM model, Process 2 would read the value of the shared variable `x` from a *universal copy* of shared memory which has not yet been touched by the executing transaction on Process 1. If Process 1 completes its transaction and commits successfully, then the final value 2 is written to this universal copy of x, and thereafter Process 2 could read this copy and thus it is possible that $r1 == 2$ after this program has finished. However, $r1 == 1$ is not possible. We note that $r1 == 1$ would be possible at the end under forms of *weak atomicity*, e.g., if `atomic` was implemented as a *synchronized* block in Java (see [13]). The middle example in Figure 1 contains an explicit abort. In the TSM model, all write operations on shared variables performed by a transaction only have an effect on (the universal copy of) shared memory if the transaction successfully commits. Otherwise they have no effect, and are not visible to anyone after the transaction has been aborted. Thus $r1 == 1$ is not possible at the end of this program. This is a form of *deferred update* as opposed to *direct update* ([18]), where writes in an aborted transaction do take effect, but the abort overwrites them with the original values. In that case, such a write might be visible to non-transactional code and `r1` might have the value 1 at the end of execution of this example. Note that our semantics for TSMs does not take into account possible re-orderings that may be performed by standard compilers or architectures. For instance, compilers are usually allowed to reorder read operations, such as those performed by Process 2 in the rightmost example in Figure 1. Such reordering issues [13] are not addressed in this paper. One could extend TSMs to incorporate notions of reordering in the model, but we feel that would complicate the model too much and detract from our main goal of having a clean abstract reference model which brings to light the salient aspects of transactional concurrent programs.

3 Definition of Transactional State Machines

In this section we define *Transactional State Machines* (TSMs). The definition resembles that of (concurrent) boolean programs and (concurrent) extended recursive state machines (see, e.g., [2,3,5]), but with additional constructs for transactions. Our definition will use some standard notions (e.g., *valuations* of variables, expressions, types, etc.) which are defined formally in the full tech report [12].

3.1 Syntax of TSMs

A *Transactional State Machine* $\mathcal{A}$ is a tuple $\mathcal{A} = \langle S, \sigma_{init}, (P_r)_{r=1}^{n} \rangle$, where S is a set of *shared* variables, σ_{init} is an initial valuation of S, and $P_1, \ldots, P_n$, are processes. Each process is given by $P_r = (L_r, \gamma^r_{init}, p_r, (A^r_i)_{i=1}^{k_r})$ where L_r is a finite set of (non-shared) *thread-local*[1] variables for process r, γ^r_{init} is an initial valu-

[1] We note that these thread-local variables are used by all procedures running on the process. For simplicity, we do not include procedure-local variables, and we assume procedures take no parameter values and pass no return values. This is done only for clarity, and we lose nothing essential by making this simplification.

ation of L_r, $p_r \in \{1, \ldots, k_r\}$ specifies the index of the *initial (main) procedure*, $A^r_{p_r}$, for process r (where runs of that process begin). The A^r_i's are the *procedures* (or *components* in the RSM terminology) for process r. We assume that the first d_r of these are *ordinary* and the remaining $k_r - d_r$ are *transactional* procedures. The two types of procedures have a very similar syntax, with the slight difference that transactional procedures have access to an additional *abort* node, ab_i. Specifically, each procedure A^r_i is formally given by: $\langle N^r_i, en^r_i, ex^r_i, ab^r_i, \delta^r_i \rangle$, whose parts we now describe (for less cluttered notation, we omit the process superscript, r, when it is clear from the context):

- A finite set N_i of nodes (which are control locations in the procedure).
- Special nodes: $en_i, ex_i \in N_i$, known respectively as the *entry node* and *exit node*, and (only for transactional components) also an *abort node* $ab_i \in N_i$.
- A set δ_i of *edges*, where an edge can be one of two forms:
  - *Internal edge:* A tuple (u, v, g, α). Here u and v are nodes in N_i, $g \in BoolExp(S \cup L_r)$ is a *guard*, given by a boolean expression over variables from $S \cup L_r$ (see the full tech report [12]). $\alpha \in Assign(S \cup L_r)$ is a (possibly simultaneous) *assignment* over these variables (again, see [12] for formal definitions). We assume that u is neither ex_i nor ab_i (because there are no edges out of the exit or abort nodes), and that v is not the entry node en_i. Intuitively, the above edge defines a possible transition that can be applied if the guard g is true, and if it is applied the simultaneous assignments are applied to all variables (all done atomically), and the local control node (i.e., program counter) changed from u to v. The set of internal edges in procedure A_i is denoted by δ^I_i.
  - *Call edge:* A tuple (u, v, g, c). u and v are nodes in N_i, $g \in BoolExp(S \cup L_r)$ is a guard, $c \in \{1, 2, \ldots, k\}$ is the index of the procedure being *called*. Again, we assume $u \notin \{ex_i \cup ab_i\}$, and $v \neq en_i$. Calls are either *transactional* or *ordinary*, depending on whether the component A_c that is called is transactional or not (i.e., whether $c > d_r$ or $c \leq d_r$). Intuitively, a call edge defines a possible transition that can be taken when its guard g is true, and the transition involved calling procedure A_c (which of course involves appropriate call stack manipulation, as we'll see). Upon returning (if at all) from the call to A_c, control resumes at control node v. The set of call edges in component i is denoted by δ^C_i.

3.2 Abort-Aware Semantics of TSMs

A full formal semantics of TSMs is given in [12] (due to space constraints). Here we give an informal description to facilitate intuition and describe salient features. TSMs model concurrent shared memory imperative procedural programs with bounded data and transactions. A *configuration* of an TSM consists of a call stack for each of the r processes, a current node (the program counter) for each process, as well as a *universal valuation* (or *universal copy*), $\mathcal{U}$, of shared variables. Crucially, during execution the "view" of shared variables may be different for different processes that are inside transactions. In particular, different processes,

when executing inside transactions, will have their own local copies (valuations) of shared variables on their call stack, and will evaluate and manipulate those valuations in the middle of transactions, rather than the single universal copy.

A transaction keeps track (on the stack) of what shared variables have been *used* and *written*. If a shared variable is written by one of the processes inside the scope of one of the transactions, the universal copy $\mathcal{U}$ is not modified. Instead, an in-scope *mutable* copy of that shared variable is modified. The *mutable* copy resides on the stack frame of the innermost transaction on the call stack for that process. The first time a shared variable is read (i.e., used) inside a transaction, unless it was already written to in the transaction, its value is copied from $\mathcal{U}$ to a *fixed* local copy for that transaction and also to a separate *mutable* local copy, both on the stack. Thereafter, both reads and writes inside the transaction will be to this mutable copy.

At the end of a transaction, the written values will either be committed or aborted. The transaction is automatically aborted if a shared memory conflict has arisen, which is checked using an atomic compare-and-swap operation as follows. We compare the values of variables in the fixed copy of shared memory with the values of those same shared variables in the universal copy $\mathcal{U}$, and if these are all equal, then for the *written* variables, we copy their valuations in the mutable copy on the stack frame to the universal copy $\mathcal{U}$. If, on the other hand, the *compare-and-swap* fails, i.e., the compared values are not all equal then we *abort* the transaction, discard any updates to shared variables, pop the transactional stack frame and restore the calling context. (How this all works is described in detail in [12]).

If we have nested transactions, and values are committed inside an inner nested transaction, then their effect will only be immediately visible in the next outer nested transaction (i.e., this follows the semantics of *closed* nested transactions), and the committed values will only be placed in the mutable copy of shared variables of the next outer transaction. Otherwise, if the inner transaction aborts, then its effect on shared variables is discarded before control returns to the calling context.

Again, see [12] for detailed semantics. We highlight here some other salient features of the semantics which will be pertinent in other discussions:

- The universal valuation $\mathcal{U}$ is only updated upon a successful commit of outermost (non-nested) transactions, not of inner (nested) transactions.
- There are two distinct ways in which an abort can occur. One is an explicit abort, which occurs if a transaction reaches a designated abort node. The other is an automatic abort, carried out by the memory system due to conflicts with universal memory. (For nested transactions, the only possible abort is an explicit one, because no conflict is possible.)

4 Correctness: Stutter-Serializability

In this section we discuss a correctness property that TSMs possess. Informally, the correctness property relates to "atomicity" and serializability of transactions,

but such notions have to be defined carefully with respect to the model. What we wish to establish is the following fact: if there exists any run π of a TSM which witnesses a (possibly infinite) sequence of changes to the universal copy (valuation) of memory, there must also exist a run π' which witnesses exactly the same sequence of changes to the universal copy, but such that all transactions which start *and which do not abort and do terminate* in π' must execute entirely serially without any interleaving of steps on other processes in the execution of the terminating transaction. Formally, this requires us to consider *stutter-invariant* temporal properties over atomic predicates that depend only on the universal valuation of shared variables in a state of the TSM, and stutter-equivalence (see [12] for definitions).

We say a run ρ of a TSM contains *only serialized successful transactions* if every transaction on any process in the run ρ that starts and successfully commits, executes serially without any interleaving of steps by other processes. In other words, the entire execution of each successful transaction occupies some contiguous sequence $\rho_i\rho_{i+1}\ldots\rho_{i+m}$ in the run. For a run ρ of $\mathcal{A}$, let $\rho[\mathcal{U}]$ denote a new word, over the alphabet of shared variable valuations, such that $\rho[\mathcal{U}]$ is obtained from ρ by retaining only the universal valuation of shared variables at every position of the run (i.e., replacing each state ρ by the universal valuation in that state). We say that a TSM, $\mathcal{A}$, is *stutter-serializable* if: for every run ρ of $\mathcal{A}$, there exists a (possibly different) run ρ' of $\mathcal{A}$ such that $\rho[\mathcal{U}]$ is stutter-equivalent to $\rho'[\mathcal{U}]$, and such that ρ' contains only serialized successful transactions.

Theorem 1. *All TSMs are stutter-serializable.*

Proof. (*Sketch)* A full proof is [12]. Here we sketch the basic intuition. If at the end of a non-nested transaction which is about to attempt to commit, the atomic compare-and-swap operation succeeds, then at exactly the point in "time" when the compare-and-swap operation executed, the values in the universal copy of shared variables used inside the transaction are exactly the same as the values that were read from the universal copy the first time these variables were encountered in the transaction. Each shared variable is read from the universal copy at most once inside any transaction. All subsequent accesses to shared variables are to the local mutable copy on the transactional stack frame. Consequently, since the values of shared variables are the only input to the transaction from its "environment" (i.e., from other processes), the entire execution of that transaction can be "delayed" and "rescheduled" in the same "infinitesimal time slot" just before the atomic compare-and-swap operation occurred, and the resulting effect of the transaction on the universal copy of memory after it commits would be identical (because it would have identical input). The only visible effect on the universal copy of memory during the run that this rescheduling has is that of adding or removing "stuttering" steps, because the rescheduled steps do not change values in the universal copy of shared memory. □

Note that TSMs can reach new states due to transactions being aborted by the run-time environment due to memory conflicts. In other words, even aborted transactions have side effects. For instance, a TSM can use a thread-local variable

to test/detect that its last (possibly nested) transaction was aborted, and take appropriate measures accordingly, including reaching new states that are reachable only following such an abort. This fact does not contradict the above correctness assertion about TSMs, because the correctness assertion does not rule out the possibility that in order for a certain feasible sequence of changes to universal memory $\mathcal{U}$ to be realized some transactions might necessarily have to abort during the run. In general, it does not seem possible to devise a reasonable model of imperative-style transactional programs where transactions that are aborted will have no side effects. Anyway, there are good reasons not to want this. One useful consequence of side effects is that one can easily implement a "retry" mechanism in TSMs which repeatedly tries to execute the transaction until it succeeds. Some transactional memory implementations offer "retry" as a separate construct (see [18]).

5 Model Checking

It can be easily observed (via arguments similar to, e.g., [23]) that model checking for general TSMs, even with 2 processes, is at least as hard as checking whether the intersection of two context-free languages is empty. We thus have:

Proposition 1. *Model checking arbitrary TSMs, even those with 2 processes, even against stutter-invariant LTL properties of shared memory is undecidable.*

On the other hand, we show next that there is an interesting class of TSMs for which model checking remains decidable. Let the class of *top-transactional* TSMs be those TSMs with the property that the *initial* (*main*) procedure for every process makes only transactional calls (but inside transactions we can execute arbitrary recursive procedures). Let us call a TSM *almost-top-transactional* if one process is entirely unrestricted, but all other processes must have main procedures which make only transactional calls, just as in the prior definition.

Theorem 2. *The model checking problem for almost-top-transactional TSMs against all stutter-invariant linear-time (LTL or ω-regular) properties of (universal) shared memory is decidable.*

Proof. Given a TSM, $\mathcal{A}$, our first task will be to compute the following information. For each process r (other than the one possible process, r', which does not have the property that all calls in its main procedure are transactional) we will compute, for every transactional procedure, A_c on process r, certain *generalized summary paths*. A *generalized summary path* (GSP) for a transactional procedure A_c is a tuple $G = (\gamma_{start}, R, \gamma_{finish}, status, \sigma)$. γ_{start} and γ_{finish} are valuations of the thread-local variables L_c. *status* is a flag that can have either the value `commit` or `abort`. σ is a *partial valuation* of shared variables, meaning it is a set of pair (x, w) where x is a shared variable and w is a value in x's domain (and there is at most one such pair in σ for every shared variable x). $R = R_1, \ldots, R_d$ is a sequence of distinct partial valuations of shared variables, where furthermore, different R_i's do not evaluate the same variable. In other

words, for each shared variable x, there is at most one pair of the form (x, w) in the entire sequence R. Such a sequence R yields a partial valuation $\sigma_R = \cup_{i=1}^{d} R_i$ (and we shall need to refer both to the sequence R and to σ_R).

We now define what it means for a GSP, G, to be *valid* for the transactional procedure A_c. Informally, this means that G summarizes one possible *terminating* behavior of the transaction A_c if it is run in sequential isolation (with no other process running). More formally, we call a GSP, $G = (\gamma_{start}, R, \gamma_{finish}, status, \sigma)$, *valid* for the transactional procedure A_c, if it satisfies the following property. Suppose a call to A_c is executed in sequential isolation (i.e., with no other process running). Suppose, furthermore that in the starting state ψ_0 in which this call is made γ_{start} is the valuation of thread-local variables L_r on process r, and that the universal copy of shared memory $\mathcal{U}$ is *consistent* with the partial valuation σ_R (in other words it agrees with σ_R on all variables evaluated in σ_R). Then there exists some sequential run of A_c from such a start state ψ_0 where during this run:

1. The sequence of reads of the universal copy of shared memory variables executed during the run corresponds precisely to the d partial valuations $R_1, \ldots, R_d$. For example, if $R_3 = \{(x_1, w_1), (x_2, w_2)\}$, then the third time during the run in which the universal copy of shared memory is accessed (i.e., third time when shared variables are used that have not been used or written before) requires a simultaneous read[2] of shared variables x_1 and x_2 from the universal copy $\mathcal{U}$, and clearly the values read will be w_1 and w_2, because $\mathcal{U}$ is by definition consistent with σ_R. (Note that $\mathcal{U}$ does not change in the middle of the sequential execution of A_c, because it is run in sequential isolation, with no other process running.)
2. After these sequences of reads, the run of A_c terminates in a state where the valuation of local variables is γ_{finish} and either commits or aborts, consistent with the value of *status*.
3. Moreover, if it does commit, then the partial valuation of shared variables that it writes to the universal copy $\mathcal{U}$ (via *compare-and-swap*) at the commit point is σ. (And otherwise, σ is by default the empty valuation.)

Let $\mathcal{G}_c$ denote the set of all valid GSPs for transactional procedure A_c. It is clear that for any transactional procedure, every GSP G is a finite piece of data, and furthermore that there are only finitely many GSPs. This is because the universal valuation of every shared variable can be read at most once during the life of the transaction, and of course there are only finitely many variables, and each variable can have only finitely many distinct values.

Lemma 1. *The set $\mathcal{G}_c$ is computable for every transactional procedure A_c.*

See [12] for a proof of the Lemma. We shall compute the set $\mathcal{G}_c$ for every transactional procedure A_c and use this information to construct a finite-state summary

[2] Again, recall that the reason there may be simultaneous reads from universal shared variables is an artifact of the strong isolation assumption combined with our formulation of (potentially simultaneous) assignment statements.

state-machine B_r, for every process r, which summarized that process's behavior. We will also describe the behavior of the single unrestricted process r' using a a Recursive State Machine (RSM), $B'_{r'}$. We shall then use these B_r's and $B'_{r'}$ to construct a new RSM $B = (\otimes_{r \neq r'} B_r) \otimes B'_{r'}$ which is an appropriate *asynchronous product* of all the B_r's and $B'_{r'}$. The RSM B essentially summarizes (up to stutter-equivalence) the behavior of the entire TSM with respect to shared memory. The construction of the B_r's, $B'_{r'}$, and B is described in [12].

It follows from the construction that B has the following properties. For every run ρ of the entire TSM, $\mathcal{A}$, there is a a run π of B such that π is stutter-equivalent to the restriction $\rho[\mathcal{U}]$ of the run ρ to its sequence of universal shared memory valuations. And likewise, for every run π of B, there is a run ρ of $\mathcal{A}$ such that $\rho[\mathcal{U}]$ is stutter-equivalent to π. (Again, see [12] for definitions pertaining to stutter-equivalence.) Thus, once the RSM B is constructed, we can use the model checking algorithm for RSMs ([2]) on B to check any given stutter-invariant LTL, or stutter-invariant ω-regular, property of universal shared memory of $\mathcal{A}$. □

We remark that the complexity of model checking can be shown to be singly-exponential in the encoding size of the TSM, under a natural encoding of TSMs. (Note that TSMs are compactly encoded: they are *extended* concurrent recursive state machines, with variables that range over bounded domains.)

Finally, we note that a similar decidability result can be obtained with other variant semantics where (1) automatic aborts are systematically considered as retries, (2) terminating transactions nondeterministically commit or abort, or (3) never more than one transaction executes concurrently (this is equivalent to the single-lock semantics). Indeed, those variant semantics are simpler to define and can be viewed as particular cases of the abort-aware TSM semantics.

6 Related Work

There is an extensive literature on Transactional Memory and there are already many prototype implementations (see the online bibliography [8], and see the recent book by Larus and Rajwar [18]). Most of this work discusses how to implement transactional memory either in hardware or software, from a systems point of view with the main emphasis on performance. Some researchers have formalized and studied the semantics of transactional memory implementations, in order to clarify subtle semantics distinctions between various implementations and the interface between these implementations and higher-level "transactional programs" running on top of them. Such distinguishing low-level semantics are quite complicated, and are not suitable for higher-level transactional program verification.

Recent work [1,20] discuss transaction semantics in the difficult setting of *weak* isolation/atomicity, where implementations do not detect conflicting accesses to shared memory between non-transactional and transactional code, and thus these may interfere unpredictably. By contrast, we assume a form of strong isolation, as described earlier. We aim for a clean model that can highlight the issues which are specific to transactions, and we do not want to obfuscate them

with difficult issues that arise by introducing weak memory models, weak consistency, out-of-order execution, and weak isolation. Such notions are somewhat orthogonal, and are problematic semantically even in settings without transactions. Our goal is to define an abstract, idealized, yet relevant, model of transactional programming that could in principle serve as a foundation for verification. There are various design choices in the implementation of a transactional memory framework (see [18] for a taxonomy of choices), and our TSM model reflects several such choices. For instance, our definition of nested transactions is a form of *closed* nested transactions. We do not consider so-called *open* nested transactions, where an inner transaction may commit while an outer one aborts (because we can not see any sensible semantics for them, even in the single-process purely sequential setting). Some of these choices are adjustable in the model, as discussed in the previous section.

Independently, [14] has recently proposed the notion of "opacity" as an alternative semantics criterion for transactions. Loosely speaking, opacity also requires serializability of aborted transactions in addition to serializability of committed transactions, with the goal of preventing aborted transactions from reading "inconsistent" values. In contrast, our abort-aware semantics does not require the stronger opacity criterion. Instead, it assumes that programmers can deal with automatically aborted transactions as they currently handle runtime exceptions in other programming languages. Of course, opacity could be formalized using an alternate TSM semantics.

Mechanisms other than transactions, such as locks, have been proposed to enforce "atomicity" and have been studied from a verification point of view. For instance, concurrent reactive programs where processes synchronize with locks were studied in [22] where a custom procedure exploiting "atomicity" (based on Lipton's reduction) is used to simplify the computation of "summaries" for such programs. Also, several verification problems are shown to be decidable in [16] for a restricted class of programs where locks are nested. Several other restrictions of concurrent pushdown processes for which verification problems are decidable have also been identified (e.g., [9], among others). There are some high-level similarity between these prior results and our results in Section 5, but the details are substantially different due to the specifics of the TSM model.

Other related work discusses how to check the correctness of implementations of transactional memory, based on lower level constructs, using testing ([19]) or model checking ([10]). By contrast, we do not address the problem of analyzing the correctness of implementations of transactional memory, but rather the correctness of transactional programs running on top of (correct) implementations.

Notions of serializability have been studied in database concurrency control for decades ([6]). However, there are subtle distinctions between the semantics of serializability in different setting. [4] systematically studied automata-based formalization of serializability and other related concepts. We formulate a clean and natural notion of *stutter-serializability* for TSMs, and show it is satisfied by them. The notion arose from our considerations of the abort-aware TSM model, and does not appear to have been studied before in the literature.

7 Conclusions

This work initiates a study of transactional programming from a program analysis and verification point of view. Our goal is to provide a formal foundation for high-level reasoning about transactional programs, which nevertheless does not ignore the meaning of manual aborts nor automatic aborts in such programs, and facilitates building program analysis and verification tools for transactional programs. In contrast with prior semantics work on transactional memory systems, we do not consider the (lower-level) verification of transactional-memory implementations but instead focus on the (higher-level) abstract semantics of transactional programs running on top of those implementations. The paper makes two main contributions.

- We propose Transactional State Machines as an abstract finite-data model for transactional programs. TSMs are essentially concurrent extended recursive state machines augmented with constructs to specify transactions. Their significant expressiveness allows the modeling of interleaved executions of concurrent and potentially nested and/or non-terminating transactions. However, we show that, provided recursion is confined to occurring inside transactions, the expressiveness of TSMs is reduced and model checking of a large class of properties becomes decidable.
- We offer a critique of the current dominant high-level semantics for transactional programming, namely the single-lock semantics, and extend it with an alternative abort-aware semantics which captures important features of real transactional programs such as explicit and automatic aborts. We identify stutter-serializability as a key formal property (enjoyed, e.g., under single-lock semantics), and we show that our abort-aware semantics still enjoys this property and provides a clean and precise high-level semantics also for explicit and automatic aborts.

TSMs are concurrent state machines so it is natural to study them under fairness assumptions that insure progress on all processes. Note that for model checking, such fairness assumptions can be specified within LTL specifications.

Acknowledgements. We thank Jim Larus for several helpful discussions and encouragements, and Martin Abadi, Tom Ball, Sebastian Burckhardt, Dave Detlefs, Tim Harris, Madan Musuvathi, Shaz Qadeer and Serdar Tasiran for helpful comments.

References

1. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of Transactional Memory and Automatic Mutual Exclusion. In: Proceedings of POPL 2008 (2008)
2. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. 27(4), 786–818 (2005)

3. Alur, R., Chaudhuri, S., Etessami, K., Madhusudan, P.: On-the-fly reachability and cycle detection for recursive state machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 61–76. Springer, Heidelberg (2005)
4. Alur, R., McMillan, K.L., Peled, D.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. 160(1-2), 167–188 (2000)
5. Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
6. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
7. Blundell, C., Lewis, E.C., Martin, M.M.K.: Subtleties of Transactional Memory Atomicity Semantics. IEEE Computer Architecture Letters 5(2) (2006)
8. Bobba, J., Rajwar, R., Hill, M. (eds.): Transactional memory bibliography (online), `http://www.cs.wisc.edu/trans-memory/biblio/index.html`
9. Bouajjani, A., Esparza, J., Touili, T.: A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In: Proceedings of POPL 2003 (2003)
10. Cohen, A., O'Leary, J.W., Pnueli, A., Tuttle, M.R., Zuck, L.D.: Verifying Correctness of Transactional Memories. In: Proceedings of FMCAD 2007 (Formal Methods in Computer-Aided Design) (2007)
11. Etessami, K.: Stutter-invariant languages, ω-automata, and temporal logic. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 236–248. Springer, Heidelberg (1999)
12. Etessami, K., Godefroid, P.: An Abort-Aware Model of Transactional Programming. Technical Report MSR-TR-2008-159, Microsoft Research (2008)
13. Grossman, D., Manson, J., Pugh, W.: What Do High-Level Memory Models Mean for Transactions? In: Memory System Performance and Correctness (MSPC 2006), pp. 62–69 (2006)
14. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proc. 13th ACM PPoPP, pp. 175–184 (2008)
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III., W.N.: Software transactional memory for dynamic-sized data structures. In: Proc. of 22nd Symp. on Principles of Distributed Computing (PODC), pp. 92–101 (2003)
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. Lamport, L.: What good is temporal logic. In: Mason, R.E.A. (ed.) Information Processing 1983: Proc. IFIP 9th World Computer Congress, pp. 657–668 (1983)
18. Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool (2007)
19. Manovit, C., Hangal, S., Chafi, H., McDonald, A., Kozyrakis, C., Olukotun, K.: Testing Implementations of Transactional Memory. In: Proceedings of the 15th international conference on Parallel architectures and compilation techniques (2007)
20. Moore, K.F., Grossman, D.: High-Level Small-Step Operational Semantics for Transactions. In: Proceedings of POPL 2008 (2008)
21. Peled, D.: Th. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. Information Processing Letters 63, 243–246 (1997)
22. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing Procedures in Concurrent Programs. In: Proceedings of POPL 2004 (2004)
23. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
24. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)

Model-Checking the Linux Virtual File System*

Andy Galloway[1], Gerald Lüttgen[1], Jan Tobias Mühlberg[1], and Radu I. Siminiceanu[2]

[1] Department of Computer Science, University of York, York YO10 5DD, UK
{andyg,luettgen,muehlber}@cs.york.ac.uk
[2] National Institute of Aerospace, Hampton, Virginia 23666-6147, USA
radu@nianet.org

Abstract. This paper presents a case study in modelling and verifying the Linux Virtual File System (VFS). Our work is set in the context of Hoare's verification grand challenge and, in particular, Joshi and Holzmann's mini-challenge to build a verifiable file system. The aim of the study is to assess the viability of retrospective verification of a VFS implementation using model-checking technology. We show how to extract an executable model of the Linux VFS implementation, validate the model by employing the simulation capabilities of SPIN, and analyse it for adherence to data integrity constraints and deadlock freedom using the SMART model checker.

1 Introduction

Hoare has proposed a 15-year grand challenge which calls on the program verification community to collaborate on building verifiable programs [16]. Joshi and Holzmann have subsequently provided a mini-challenge [19] of building a verifiable file system as a stepping stone towards meeting Hoare's challenge. Neither challenge overly constrains the verification approach. On the one hand, for example, there is the *constructive* approach in which formal reasoning is employed to first establish the validity of a specification and then the correctness of an implementation with respect to the specification. On the other hand, the *analytical* approach aims to build a valid abstract model of an existing implementation and then to show that this model satisfies some correctness criteria.

This paper applies the analytical approach to verifying an implementation of the *Virtual File System* (VFS) layer [4] within the Linux kernel, using model-checking technology. This layer is of particular interest since it provides support for implementing concrete file systems such as EXT3 and ReiserFS, and encapsulates the details on top of which C POSIX libraries are defined; such libraries in turn provide functions, e.g., *open* and *remove*, that facilitate file access (cf. Sec. 2). The aim of our case study is to assess the feasibility of analytical program verification to Joshi and Holzmann's mini-challenge. In particular, we

* This research was partially funded by EPSRC under grant GR/S86211/01 and by NASA under cooperative agreement NCC-1-02043.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 74–88, 2009.

are interested in whether and how an appropriate model of the VFS implementation can be constructed, and if meaningful verification results can be obtained given the limitations of current model-checking technology.

Our first contribution is in modelling the complex Linux VFS implementation that consists of more than 65k lines of C code, based on an analysis of the data structures and algorithms employed in VFS (cf. Sec. 3). Despite the recent advances in software model checking — as exemplified by the BLAST model checker [15] and Microsoft's Static Device Verifier (*www.microsoft.com/whdc/devtools/tools/SDV.mspx*) which, under the hood, automatically extract models from C code —, one quickly reaches their limits when applying them to operating systems code, such as the VFS code. This is because such code makes use of dynamic memory allocation, function pointers, macros and inlined assembly [21]. Until techniques addressing these shortcomings have matured, building models from operating systems code remains largely a manual task.

Our VFS model is the result of slicing away and abstracting some details of the VFS data structures and algorithms. This is done in a way that makes the model amenable to modern model checkers while maintaining all details necessary for checking non-trivial data-integrity properties. The model is expressed abstractly in a subset of C, so that it can easily be reused by others. While building the model took several weeks, its validation via reviewing and simulation consumed several months. The simulation was carried out in the SPIN model checker [17] since SPIN has rich simulation capabilities, with support for run-time assertions, and an input language into which our model can be cast straightforwardly. However, since our model is sufficiently close to the VFS implementation and thus exhibits a large state space with wide state vectors, we were unable to run SPIN in verification mode, even when disallowing concurrent access to VFS functions. Also, the VFS model cannot be verified by model checkers that do not support concurrency, such as BLAST.

The paper's second contribution is the formal verification of our VFS model by using model checking to analyse low-probability scenarios, thereby increasing confidence in the correctness of the Linux kernel (cf. Sec. 4). In particular, we were looking for, and not expected to find, the corruption of the underlying data state and deadlocks. The challenge here is to identify data-integrity properties from the rather shallow VFS documentation. To conduct the verification, we translated our model into Petri nets and used the model checker SMART [6] which implements efficient, decision-diagram-based algorithms for analysing concurrent systems. SMART was chosen here because of our familiarity with the tool and its proven record for analysing complex models, including NASA's Runway Safety Monitor [22] and the SPIDER clock synchronisation and self-stabilisation protocols [20]. While the VFS model pushes SMART to its limits, we were able to successfully prove all considered properties.

Our case study is novel because of its approach and scope. It tests the feasibility of reverse-engineering a model of an existing file system, including data structures and locking mechanisms, and of checking such a model for adherence to healthiness properties. This contrasts with other work in the field (cf. Sec. 5)

which employs either the constructive approach to verification [2, 11], or model-checking as a run-time verification technique for driving a test-harness for the implementation [23, 24]. Our file system model is of particular interest to NASA which is currently developing, together with JPL, a pilot project to help build a reliable file system for flash memory.

2 The Linux Virtual File System

This section introduces the Linux file system architecture and, in particular, the Virtual File System layer. For a more detailed description, we refer the reader to [4] and *www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html*. An overview of the VFS internals and data structures is presented in Fig. 1.

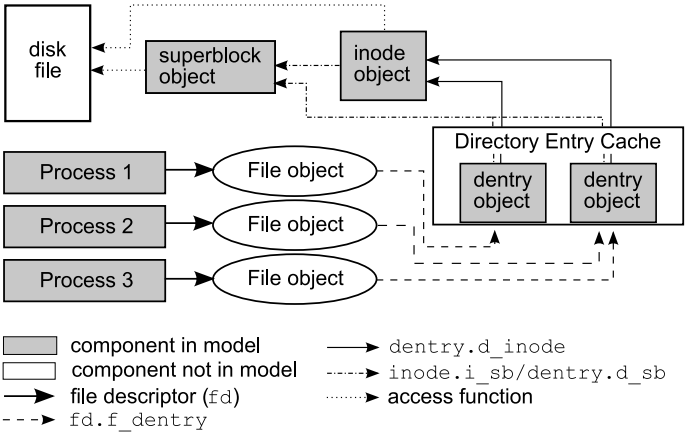


Fig. 1. Illustration of the VFS environment and data structures

Architecture. The Linux file system architecture consists of six layers. The most abstract is the *application* layer which refers to the user programs; this is shown as Process 1 to 3 in Fig. 1. Its functionality is constructed on top of the file access mechanisms offered by the *C POSIX library*, which provides functions facilitating file access as defined by the POSIX Standard [14], e.g., open `open()`, delete `remove()`, make directory `mkdir()` and remove directory `rmdir()`. The next lower layer is the *system call interface* which propagates requests for system resources from applications in user space to the operating system kernel.

The *Virtual File System* layer is an indirection layer, providing the data structures and interfaces needed for system calls related to a standard Unix file system. It defines a common interface that allows many kinds of specific file systems to coexist, and enables the default processing needed to maintain the internal representation of a file system. The VFS runs in a highly concurrent environment as its interface functions may be invoked by multiple, concurrently executing application programs. Therefore, mechanisms implementing mutual exclusion are

widely used to prevent inconsistencies in VFS data structures, such as atomic values, mutexes, reader-writer semaphores and spinlocks. In addition, several global locks are employed to protect the global lists of data structures while entries are appended or removed. To serve a single system call, typically multiple locks have to be obtained and released in the right order. Failing to do so could drive the VFS into a deadlock or an undefined state.

Each *specific file system*, such as EXT2, EXT3 and ReiserFS, then implements the processing for supporting the file system and operates on the data structures of the VFS layer. Its purpose is to provide an interface between the internal view of the file system and physical media by translating between the VFS data structures and their on-disk representations. Finally, the lowest layer contains *device drivers* which implement access control for physical media.

Data structures. The most relevant data structures are *superblocks*, *dentries* and *inodes*, whose names are used in different contexts outside the VFS; we employ the VFS-related definitions rather than their file-system-specific meanings or their on-disk representations (cf. Fig. 1). The *super_block* data structure describes the abstract properties of the file system, such as its type (e.g., EXT3), the physical device on which it resides, its total size, its mount point and a pointer to the root dentry. The `struct super_block` is defined in `include/linux/fs.h`.

The *dentry* data structures collectively describe the structure of the file system. Each dentry contains a file's name, a link to the dentry's parent, the list of subdirectories and siblings, hard link information, mount information, a link to the relevant super block, and locking structures. It also carries a reference to its corresponding inode, and a reference count that reflects the number of processes currently using the dentry. Dentries are hashed to speed up access; the hashed dentries are referred to as the Directory Entry Cache, or *dcache*, which is frequently consulted when resolving path names embedded within function calls. The `dentry` struct is defined in `include/linux/dcache.h`.

The *inode* data structure carries information specific to a file, whether it is a regular file, directory or device. This includes a link to the relevant super block, backward links to the dentries referencing the inode, file permissions, file type, file size, operations for use on inodes by the VFS, callbacks to the specific file system, device-specific information, and information about how the file is memory-mapped, e.g., it links to file objects which capture the data needed to support file descriptors in user space. The struct `inode` is defined in `include/linux/fs.h`.

Implementation. The public interface of the Linux 2.6.18 VFS consists mainly of the header files `fs.h`, `namei.h` and `dcache.h` residing in `include/linux`. The implementation of system calls can be found in the `fs` subdirectory of the kernel source tree. Here, the files `dcache.c`, `namei.c`, `inode.c`, `stat.c` and `open.c` are notable; they contain the logic for the system calls featured in our model.

To explain the interaction between the different parts of the VFS, we take the `creat()` system call as an example. The functions involved comprise roughly 5k lines of source code, not including data structure definitions and macro expansions. In POSIX, the signature of `creat()` is defined as:

```
int creat(const char *pathname, mode_t mode);
```

providing the full path to the file to be created and the desired file permissions. In the following we discard all permission handling. The VFS entry point for `creat()` is the function `sys_creat()` defined in `open.c` which redirects to

```
sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

Therefore, `creat()` is handled as a special case of the `open()` system call. `sys_open()` then triggers `do_sys_open()` that calls `do_filp_open()`, which in turn invokes `open_namei()`. This last function resides in `namei.c` and represents the main part of the `open` routine. It first uses `do_path_lookup()` to traverse the dentry directory tree. This involves increasing and decreasing usage counters by calling `dget()`/`dput()` from `dcache.c` and obtaining locks for dentries belonging to the path. If at least the parent directory of the file to be created exists, `do_path_lookup()` returns successfully, passing a pointer to the parent's dentry. If the target file for the `creat()` operation does not yet exist, the path lookup function will return a dentry that is not yet associated with an inode. In that case, `open_namei()` invokes `vfs_creat()` to propagate the creation of an inode down to the specific file system and link the newly created inode to the dentry. At this point, the file creation is complete.

Additionally, the process of creating a file involves obtaining and releasing several reader-writer semaphores as well as the `i_mutex` of the parent's inode. It also has to obtain global spinlocks protecting complete lists of dentries and inodes, in case the execution of the system call is preempted by the scheduler.

Properties. One aim of our study is to show the absence of two principal types of error in the VFS implementation. Firstly, since locking mechanisms are a key part of the implementation, we are interested in the existence of scenarios that might cause *deadlock*. Secondly, since some fields of the data structures are logically dependent, we wish to establish *integrity* properties implying that the file system is being maintained in a consistent state. These properties encapsulate relationships between the data structures that ought to hold universally or between calls to VFS functions. These integrity properties are of three types:

(a) *Allocation properties*, expressing that the information pointed to by the fields of assigned nodes is allocated;
(b) *Reference properties*, expressing that reference counters are maintained in a healthy way by the functions operating on the VFS;
(c) *Structural properties*, expressing static relationships between the data structures of the VFS, which ought to be maintained by the functions operating on these structures.

3 Extracting and Validating Our VFS Model

This section presents our model of the Linux VFS implementation, discusses our adopted methodology for extracting the model, justifies our key modelling decisions, and summarises our approach to validating the model.

Initial considerations. The Linux VFS implementation is very large in size: approximately 65k lines of C code are directly relevant to the VFS and must be analysed thoroughly, whilst roughly 80k lines of code which include details of, e.g., memory and scheduling, are less directly relevant but require consideration nonetheless. Concurrency mechanisms and the use of macros also add to the complexity. Therefore, automation of the modelling process is a key concern.

To this end, we initially explored Modex [18] which can be used to extract high-level SPIN models directly from C source code. The aim was to use Modex in the initial phase of modelling to produce a slice of the VFS implementation. Unfortunately, despite attempts to simplify the task by, e.g., preprocessing the source code, Modex failed to parse the kernel source. This was likely due to non-standard and compiler-dependent source code fragments. We also considered software model checkers, such as BLAST [15], but found them to be ineffective for similar reasons. In the end, the only recourse was to use automation less directly, to support an essentially manual modelling process.

To facilitate this, we needed to carefully analyse the VFS implementation to identify the data structures that have to be captured by the model, and the integrity properties that these structures ought to satisfy. Despite the wealth of available information on the Linux VFS, these usually comprise English language descriptions of the data structures together with associated operation signatures. In contrast, [11] provides a formal specification of part of the POSIX interface. However, this is too abstract for our purposes as it avoids VFS-level considerations such as the maintenance of internal data structures and how locking mechanisms are employed. Because we required precision in order to produce an accurate model, our model had to be derived from the source code itself.

Modelling decisions. Since our case study is embedded within a research project, the scope of our VFS model had to be adjusted so as to fit the project's schedule. Therefore, we chose to incorporate only the basic operations on objects in the file system: creating files and directories, and deleting files and directories. Other POSIX commands, e.g., regarding mounting and links, were deferred; this means that only a single superblock structure is necessary. In addition, we treat files as atomic entities, thus abstracting from file content.

For practical purposes, it is necessary to impose a limit on the size of the file system that the model is to maintain. To keep state spaces tractable, we set this limit to eight nodes including root. The choice of eight nodes means that we were able to investigate operations on non-trivial configurations of the file system whilst remaining within the bounds imposed by current model-checking technology. In particular, we avoided modelling the dentry hash table as it is an unnecessary cost given the eight-nodes limit; the hash table look-up function can instead be modelled efficiently as a search over all dentries.

A final modelling decision underlying this case study embraces a dynamic file structure where links are explicit, rather than a fixed structure where links are implicit and inferable from each node; this means that we were able to remain more faithful to the implementation.

Representing the VFS data structures. Central to the VFS implementation are the logical structures of the file system (including parents, siblings and subdirectories) and the locking mechanisms (including spinlocks, mutexes and reader-writer semaphores). Consequently, the superblock, dentry and inode data structures of the VFS implementation must be represented in our model, and the most significant issue becomes which of the structures' fields should be included, how to represent them, and which fields can be omitted.

The process of identifying the fields of interest must be based on the information contained in the header files and consider how each field is used by the VFS implementation. Full details of our engagement in this lengthy process can be found in a technical report [13]. For example, for the dentry table, the key fields are: (i) *d_lock*, since locking is essential for concurrency; (ii) *d_inode*, *d_parent*, *d_child* and *d_subdirs*, since these capture the structure of the file system; (iii) *d_count*, which records whether a dentry is assigned and the number of processes accessing the dentry. Additionally, an *is_allocated* boolean flag was introduced to model dynamic data allocation.

The next decision is how to represent each field and estimate the number of bits required. Our parameters for the *dentry* table are: (i) *d_lock* is assigned three bits: one for the status of the lock, one for the process holding the lock (up to two processes), and one indicating a waiting process; (ii) *d_inode* and *d_parent* are assigned three bits each, allowing one to reference a maximum of eight inodes and dentries, whereas *d_child* and *d_subdirs* are allocated eight bits each, allowing up to eight siblings and children of a dentry to be marked rather than stored as a linked list; (iii) *d_count* is allocated three bits, permitting up to two processes to access a dentry at a time, with one bit contingency. In addition, *d_iname* is allocated three bits, allowing for eight different names and giving a maximum directory structure width of seven (plus the root).

Extracting the VFS model. The aim of the extraction process was to isolate the algorithms operating on the identified VFS data structures, and to express these in C syntax in an abstract way. The choice of using C as the modelling language also simplified the validation of our model, since it eases comparisons between model and implementation and since it can easily be fed into simulators.

As indicated earlier, the task of extracting a model from the VFS implementation was made difficult by a number of factors, including the size of the implementation and the heavy use of dynamic memory allocation and function pointers. Concurrency issues also contributed to the complexity of the exercise. To provide at least some automated support for the task, we generated call traces from kernel executions, which allowed us to obtain a series of algorithmic "snapshots" and thus an accurate impression of functionality and ordering. For example, by analysing the traces for `sys_creat()`, it was possible to confirm its behaviour that we presented in Sec. 2.

To obtain traces from a running Linux kernel we adopted the *Kernel Function Trace* tool (KFT, *tree.celinuxforum.org/CelfPubWiki/KernelFunction*) to work with Linux 2.6.18, implemented a few simple test drivers that initialised KFT for a particular system call such as `sys_creat()`, executed the call and

obtained the trace. KFT itself employs the *finstrument-functions* (*gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.htm*) capability of the compiler to add instrumentation call-outs to every function entry and exit, which are used to dump the jump and return addresses to a trace log. With the help of the kernel's symbol table, the log entries can be translated into their respective function names. However, the view of the VFS we obtained from call traces is necessarily incomplete, and a great deal of effort still had to be spent manually inspecting the code. This is due to several reasons: (i) call traces do not reveal how a particular function operates on the VFS data structures of interest; (ii) macros are not instrumented; (iii) several important function calls are missing in each trace since some functions cannot be instrumented; this is because they have to be called from an atomic context in which performing blocking I/O operations, i.e., writing out a log message, is not permitted.

Using call traces and manual inspection we were able to model the core behaviour of the VFS within several person weeks. Table 1 presents the model fragment which we extracted for the `creat()` function discussed above. Similar fragments were produced for the system calls `sys_unlink()`, `sys_mkdir()`, `sys_rmdir()` and `sys_rename()`, for various additional VFS functions such as `path_lookup()` and `path_release()`, as well as for functions that belong to other parts of the kernel's infrastructure. Due to space constraints — the complete model is about 3k lines — we cannot show it in full here. However, the final model can be downloaded from *research.nianet.org/~radu/VFS/*.

Validating the VFS model. In the absence of full automation, we adapted two classic techniques for validation: (a) our final model was extensively *reviewed* and cross-checked against the implementation, with an overall effort of two person months; (b) a similar effort was spent in *simulating* our model.

For conducting the simulation runs, we employed the SPIN verifier [17] for two reasons. Firstly, the syntax of SPIN's input language Promela is close to the C syntax adopted by our model. Therefore, the translation could be performed quickly and with little risk of introducing errors. Secondly, SPIN's rich simulation capabilities, along with the ability to add assertions, allowed for a rigorous testing regime to be implemented. To aid simulation, our SPIN model was confined to a single process, thereby eliminating the complexity introduced by concurrency. More than 100 different simulation runs were conducted on the SPIN model that was heavily injected with assertions; about 3% of the model's lines are assertion statements. Each run was performed as a simulation of one of the system calls from a recognised 'healthy' state, involving creating/deleting existing/non-existing files/directories at various levels, attempting to delete the root and copying a directory onto itself, etc. Several early errors in our VFS model were identified and corrected by these means.

Part of the model validation was also carried out during the verification phase, which involved the model checker SMART [6] as described below. Indeed, several errors were eliminated as part of the SMART modelling process where, in the first instance, model discrepancies such as unexpected verification results, property violations and deadlocks were treated as potential signs of an invalid

Table 1. Model fragment for sys_creat()

```
int sys_creat (string path) {
 lookup_res_t l;
 inode_t itmp;
 dentry_t parent, file;

 l = path_lookup (path);
 parent = *l.parent;
 file = *l.file;
 if (!parent.is_allocated)
  {
   if (file.is_allocated)
    /* deals with root look up */
    { dput(file); }
   return (ERROR);
   }

 down (parent.d_inode->i_mutex);

 if (file.is_allocated &&
      !is_directory (file))
  { up (parent.d_inode->i_mutex);
    path_release (file);
    return (SUCCESS); }

 if (file.is_allocated &&
    is_directory (file))
  { up (parent.d_inode->i_mutex);
    path_release (file);
    return (ERROR); }

 spin_lock (dcache_lock);
 file = allocate_dentry(
   last_component(path), parent);
 if (!file.is_allocated)
  { spin_unlock (dcache_lock);
    up (parent.d_inode->i_mutex);
    dput (parent);
    return (ERROR); }
 dget (file);

 spin_lock (inode_lock);
 itmp = allocate_inode(file);
 file.d_inode = &itmp;
 spin_unlock (inode_lock);
 if (!file.d_inode->is_allocated)
  { atomic_write (file.d_count, 0);
    dput (parent);
    spin_unlock (dcache_lock);
    up (parent.d_inode->i_mutex);
    return (ERROR); }

 update_parent(
    *((dentry_t *)file.d_parent));
 path_release (file);
 spin_unlock (dcache_lock);

 up (parent.d_inode->i_mutex);

 return (SUCCESS);
}
```

abstraction. For each discrepancy, the VFS model was re-checked against the VFS implementation and, if appropriate, revised.

4 Verifying Our VFS Model Using SMART

The next step in our case study was to verify our validated VFS model for absence of deadlock and adherence to data-integrity constraints using model-checking technology. To do so, we initially attempted to run SPIN in verification mode on the (single-process) SPIN model and established freedom from assertion violations. However, the analysis on a modern PC failed for all but the most trivial configurations that involve two or three nodes only, since the sizes of the state vector and the reachable state spaces are simply too large to be represented explicitly, even using advanced features such as collapse compression.

Also model checkers such as BLAST [15] cannot deal with our VFS model, due to the presence of concurrency in the VFS environment.

We therefore shifted our focus to SMART, which is a state-of-the-art symbolic model-checker for concurrent systems [6] with which we are most familiar. SMART implements the *Saturation* algorithm [8] which exploits properties of interleaving semantics for manipulating decision diagrams and can be significantly more time- and memory-efficient than SPIN [7]. As SMART provides a notation based on Petri nets rather than a software modelling language, we had to rephrase our VFS model into a Petri net. This involved introducing a program counter and circumventing the unavailability of advanced (and recursive) data structures. In addition, our SMART model had to comply with a restriction imposed by Saturation which, informally, demands that Petri net places that are functionally dependent on others be grouped in the same net partition [8].

Our VFS model as a Petri net. As for the VFS model in C, the SMART model is parameterised by the maximum number of dentries (ND), inodes (NI), and concurrent processes (NP). The encoding employed for translating the VFS model to SMART is rather straightforward: variables are represented as Petri net places, and instructions are represented as Petri net transitions. Moreover, the fields of the dentry and inode data structures are represented as arrays, i.e., the *d_parent* field of dentry k is the element $d\_parent[k]$.

In constructing the SMART model, the VFS algorithms and data structures were abstracted in a number of minor ways to make it possible to capture the required behaviour without introducing incidental complexity. One example is the need to represent path arguments to system calls and the traversal of the dentry tree structure. Lists, as used in VFS, are not native to the SMART language, and introducing them artificially would have incurred unacceptable overheads. Instead, our SMART model indexes the fully qualified filenames present in the system with natural numbers. This means that the *d_iname* field could be dropped, which also simplifies the `path_lookup()` function.

Another aspect involves the deallocation of unused nodes, which in the VFS implementation is performed separately by a garbage collection process. Our SMART model assumes an "as early as possible" deallocation in order to sequentialise deallocation and minimise complexity. Further abstractions concern the `d_subdirs` field that is used to record whether the dentry is a directory, rather than the identity of its sub-directories, and the `d_child` field that is used to record the number of siblings, rather than the identity of the siblings.

Again, the resulting VFS model cannot be reproduced here due to space constraints — e.g., the SMART code for the `creat()` function alone is 650 lines —, but is available for download from *research.nianet.org/~radu/VFS/*. The VFS model ranks with the most complex systems ever modelled in SMART. This perspective is not only reflected by the sheer size of the model (2,900 lines of SMART code), but also by the inherent complexity of the VFS. For comparison, two other similar industrial-size applications modelled in SMART are *NASA's Runway Safety Monitor* [22] (1,850 lines) and *NASA's clock synchronisation and self-stabilisation protocols* [20] for the SPIDER architecture (1,190 lines).

Integrity properties. As stated in Sec. 2, we wished to verify that the VFS model is free of deadlock and that it maintains integrity properties on its data structures. For the latter, we concentrated on the following three properties:

(a) If a node is assigned, then its parent is allocated. Here, 'assigned' means that the node itself is allocated and marked as in use (i.e., $d\_count > 0$). This is an allocation property that may be formalised by:

$$\forall d1, d2 : Dentry \bullet\ d1.is\_allocated{>}0 \wedge d1.d\_count{>}0 \wedge d1.d\_parent{=}d2 \Rightarrow d2.is\_allocated{>}0.$$

(b) When the system is stable, i.e., between file system operations, all allocated nodes' d_counts are either 0 or 1. This means that a node's reference counter does not imply that the node is in the process of inspection or alteration between operations. This is a reference property that may be formalised by:

$$\forall d : Dentry \bullet\ stable \wedge d.is\_allocated{>}0 \Rightarrow d.d\_count{=}0 \vee d.d\_count{=}1,$$

where predicate *stable* is defined using the value of the program counter.

(c) The only cycle in the parent relation is the one on root. (By default, the parent of the root is itself.) This is a structural property that, for a file system of at most eight nodes, may be formalised by:

$$\forall d1, d2, d3, d4, d5, d6, d7, d8 : Dentry \bullet$$
$$d1.is\_allocated{>}0 \wedge d1.d\_count{>}0 \wedge \ldots \wedge d8.is\_allocated{>}0 \wedge d8.d\_count{>}0 \wedge$$
$$d1.d\_parent{=}d2 \wedge \ldots \wedge d7.d\_parent{=}d8 \Rightarrow d8{=}root.$$

Formulating the required properties in SMART, including deadlock freedom, amounts to re-expressing them as simple operations over decision diagrams. This is straightforward except for the cycle-freedom property which we capture as a set of properties: "no cycles of length one, except for root"; "no cycles of length two"; "no cycles of length three", etc.

Verification results. Before verifying the properties of interest, it was necessary to construct the state space of the model. Various instantiations of ND, NI and NP were examined. Table 2 lists the results when conducting our experiments using the 64bit version of SMART on a 3.2GHz machine with 8GB of memory running Redhat Enterprise Linux version 2.6.9-5ELsmp. The most significant

Table 2. State-space generation results for SMART

ND	NI	states	time (sec)	mem. (MB)	ND	NI	states	time (sec)	mem. (MB)
1 process					2 processes				
2	2	325	1.02	1	2	2	222,715	258.49	223
3	3	12,077	9.94	12	2	3	222,715	318.77	233
4	4	1,085,247	77.27	131	3	2	-	-	>8,000
5	5	173,247,829	1,056.88	2,147	3	3	-	-	>8,000

contributor to the complexity is the number of concurrent processes, with NP≥3 unanalysable, and NP=2 with ND>2 also exceeding memory.[3]

The integrity properties formalised above were checked successfully against the generated state space, for each instantiation of ND, NI, and NP. Additionally, we verified the following properties: "root is always allocated"; "root is always in use", i.e., $d\_count>0$ is an invariant; and "the parent of an assigned node is in use". Each property was checked for each configuration in negligible time of less than one second, and shown to hold. Collectively the properties imply that every node currently in use is connected to the root.

We also checked each configuration for deadlocks. Initially, the model contained deadlocked states, for the truly concurrent setting (NP=2). Further analysis revealed that this was due to the implementation's critical use of a structure that had been abstracted away, the *dcache*. An extra bit was added to the *Dentry* structure to represent the missing information, and the model was revised accordingly. The model was then shown to be deadlock free, taking negligible time of less than one second for each configuration.

A livelock scenario was also uncovered by SMART when two processes attempt to `unlink()` the same file simultaneously. After an extensive analysis of the source code, this was attributed to the abstraction of protocols and scheduling policy designed to ensure fairness over the way spinlocks were accessed. Unfortunately, little documentation exists on the actual implementation of the scheduler for us to be able to give a firm verdict on whether the scenario is a false positive. However, this shows that modern model checkers can check more than safety properties on our VFS model. Indeed, from a model-checker's standpoint, we investigated three categories of properties: (i) safety properties which, once the reachable state space is constructed, require a single decision-diagram operation; (ii) deadlock which requires a single backward image computation on the reachable state space; (iii) livelock which requires a fixed-point computation.

5 Related Work

While [9, 10] provide techniques for verifying the correct use of file system interfaces represented as finite-state machines, work on verifying properties of file system implementations is relatively scarce.

An ongoing research project on verifying a POSIX-compliant file store, from the application interface down to the data representation on a physical medium, is outlined in [12]. Current work focuses on the construction of formal models of NAND flash memory and the commands that are used to operate it.

A correctness proof for a basic file system with standard data structures and fixed-sized disk blocks is presented in [2]. It uses the Athena theorem prover and employs the constructive approach to verification (as does [11]). The Athena

[3] The state spaces for two processes and ND=2 are indeed identical for NI=2 and NI=3. This is because the third inode is never used in this configuration, since ND<NI and because the allocation policy always returns the first available index.

model involves some of the data structures covered here and also their respective media representation. The work differs from ours in that it does not deal with concurrency-related issues and does not consider a 'real' file system implementation, thereby avoiding the process of model extraction.

Actual file system implementations are studied by Engler et al. in [23, 24]. In [24], model checking is used within the systematic testing of EXT3, JFS and ReiserFS. The verification system consists of an explicit-state model checker running the Linux kernel, a file system test driver, a permutation checker which verifies that a file system can always recover, and a recovery checker using the *fsck* recovery tool. The system starts with an empty file system and recursively generates successive states by executing system calls affecting the file system. After each step, the system is interrupted and *fsck* is used to check whether the file system can recover to a valid state. This approach is combined in [23] with symbolic execution for generating pathological test cases. In contrast to our work, [23, 24] employ run-time verification techniques that cannot exhaustively explore the implementation's state space. However, an advantage over our work is that these techniques do not require manual model extraction.

Verification approaches that model-check the source code of operating system components are presented in [3, 5, 15]. In theory, these are able to prove a file system implementation to be, e.g., free of deadlock. However, as shown in [21], the model checkers employed in [3, 5, 15] also require manual preprocessing of source code. An approach to verifying the implementation of a microkernel's paging mechanism, including a hard disk driver implemented in a fully formalised subset of C and inline assembly, is presented in [1].

6 Conclusions and Future Work

In response to Joshi and Holzmann's mini challenge, we have constructed and verified a small model of several key components of the Linux Virtual File System (VFS). This proved to be a challenging task since current automated techniques for extracting models from C source code cannot deal with important aspects of operating systems code, including macros, dynamic memory allocation, architecture- and compiler-specific code, and inlined assembly. Extracting our model by hand was made especially difficult and time-consuming by the VFS implementation's concurrency mechanisms, uncommon coding styles, and the sheer volume of code. Much time was spent in validating our model via reviews and simulation runs in SPIN. Using the SMART model checker, this model was then shown to respect data-integrity properties and to be deadlock free. The three variants of our VFS model, in C syntax, SPIN's Promela language and SMART's Petri nets, are available for download from *research.nianet.org/~radu/VFS/*.

Our case study clearly demonstrates the feasibility of abstracting data structures and algorithms from a complex file-system implementation, analysing their behaviour via simulation and model checking, and inferring conclusions about the implementation's correctness. However, automated extraction of faithful models is paramount in analytical software verification and must be a continuing

focus for research. This must involve not just program slicing but also representational changes in data structures and algorithms.

Some take-away messages. Here is what this VFS case study has taught us personally, in general terms and regarding various aspects of our work:

Goal: It makes a big difference whether one targets "bug discovery" (debugging) or "bug absence" (verification).

Automation: There is a stringent need for automating model extraction, but no existing tool is mature enough to have served our purpose.

Soundness: Building multiple models is important for fully understanding the underlying system; however, our 'staged' approach could be strengthened by checking the links between the stages formally.

Complexity: Certain aspects of the system, such as the operating system's scheduler which is external to the VFS code, cannot be faithfully modeled without dramatically increasing the size and complexity of the model.

Scalability: The fact that modern model checkers cannot handle larger parameters of our model should not be seen as a deterant since model checking technology is improving quickly.

Future work. It would be valuable to extend the scope of our case study. For instance, considering more functionality such as the specific file system layer would enable more direct comparisons with other approaches to the mini-challenge, e.g., [12]. An alternative way to extend the scope would be to incorporate an abstract model of the scheduler which, e.g., would allow one to adequately check for the absence of livelocks.

Acknowledgments. We thank the reviewers for their insightful comments, and in particular for suggesting the inclusion of 'take-away' messages.

References

[1] Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)

[2] Arkoudas, K., Zee, K., Kuncak, V., Rinard, M.C.: Verifying a file system implementation. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 373–390. Springer, Heidelberg (2004)

[3] 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)

[4] Bovet, D.P., Cesati, M.: Understanding the Linux Kernel. O'Reilly, Sebastopol (2002)

[5] Chaki, S., Clarke, E.M., Groce, A., Ouaknine, J., Strichman, O., Yorav, K.: Efficient verification of sequential and concurrent C programs. FMSD 25(2-3), 129–166 (2004)

[6] Ciardo, G., Jones III, R.L., Miner, A.S., Siminiceanu, R.: Logic and stochastic modeling with SMART. Performance Evaluation 63(6), 578–608 (2006)

[7] Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state-space generation. FMSD 31(1), 63–100 (2007)
[8] Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 328–342. Springer, Heidelberg (2001)
[9] Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI, pp. 57–68. ACM, New York (2002)
[10] DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: PLDI, pp. 59–69. ACM Press, New York (2001)
[11] Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/EVES: An experiment in the verified software repository. In: ICECCS, pp. 3–14. IEEE, Los Alamitos (2007)
[12] Freitas, L., Woodcock, J., Butterfield, A.: POSIX and the verification grand challenge: A roadmap. In: ICECCS, pp. 153–162. IEEE, Los Alamitos (2008)
[13] Galloway, A., Mühlberg, J.T., Siminiceanu, R., Lütgen, G.: Model-checking part of a Linux file system. Technical Report YCS-2007-423, U. of York, UK (2007), `www.cs.york.ac.uk/ftpdir/reports/YCS-2007-423.pdf`
[14] The Open Group. The POSIX 1003.1, Edition Specification (2003)
[15] Henzinger, T., Jhala, R., Majumdar, R., Necula, G., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
[16] Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50(1), 63–69 (2003)
[17] Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003)
[18] Holzmann, G.J., Smith, M.H.: Software model checking – Extracting verification models from source code. In: FMPEDS, pp. 481–497. Kluwer, Dordrecht (1999)
[19] Joshi, R., Holzmann, G.J.: A mini challenge: Build a verifiable filesystem. Formal Aspects of Computing 19(2), 269–272 (2007)
[20] Malekpour, M.R.: A Byzantine fault-tolerant self-stabilizing protocol for distributed clock synchronization systems. Technical Report TM-2006-214322, NASA Langley Research Center (2007)
[21] Mühlberg, J.T., Lüttgen, G.: Blasting Linux code. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007)
[22] Siminiceanu, R., Ciardo, G.: Formal verification of the NASA Runway Safety Monitor. STTT 9(1), 63–76 (2007)
[23] Yang, J., Sar, C., Twohey, P., Cadar, C., Engler, D.R.: Automatically generating malicious disks using symbolic execution. In: Security and Privacy, pp. 243–257. IEEE, Los Alamitos (2006)
[24] Yang, J., Twohey, P., Engler, D.R., Musuvathi, M.: Using model checking to find serious file system errors. In: OSDI, pp. 273–288. USENIX (2004)

LTL Generalized Model Checking Revisited

Patrice Godefroid[1] and Nir Piterman[2,⋆]

[1] Microsoft Research
[2] Imperial College London

Abstract. Given a 3-valued abstraction of a program (possibly generated using static program analysis and predicate abstraction) and a temporal logic formula, generalized model checking (GMC) checks whether there exists a concretization of that abstraction that satisfies the formula. In this paper, we revisit generalized model checking for linear time (LTL) properties. First, we show that LTL GMC is 2EXPTIME-complete in the size of the formula and polynomial in the model, where the degree of the polynomial depends on the formula, instead of EXPTIME-complete and quadratic as previously believed. The standard definition of GMC depends on a definition of concretization which is tailored for branching-time model checking. We then study a simpler *linear completeness preorder* for relating program abstractions. We show that LTL GMC with this weaker preorder is only EXPSPACE-complete in the size of the formula, and can be solved in linear time and logarithmic space in the size of the model. Finally, we identify classes of formulas for which the model complexity of standard GMC is reduced.

1 Introduction

Generalized model checking [BG00] is a way to improve precision when reasoning about partially defined systems. Such systems can be modeled as 3-valued Kripke structures where atomic propositions are either $true$, $false$ or *unknown*, denoted by the third value $\perp$. Three-valued models are a natural representation of program abstractions generated automatically [GHJ01, GWC06] using static program analysis and predicate abstraction [GS97] for software model checking [BR01].

Given a 3-valued model M and a temporal-logic formula ϕ, the generalized model-checking problem is to decide whether there exists a complete system M' that is consistent with M and that satisfies the formula ϕ. From a practical point of view, generalized model checking (GMC) can sometimes [GH05, GC05] improve verification of program abstractions. From a theoretical point of view, studying GMC is arguably interesting in its own right since GMC generalizes both model checking (when all proposition values in the model are known) and satisfiability checking (when all proposition values are unknown), probably the two most studied problems related to temporal logic and verification.

In this paper, we revisit GMC for *linear-time temporal-logic* (LTL) formulas. First, we show that LTL GMC is 2EXPTIME-complete in the size of the formula and polynomial in the model, where the degree of the polynomial depends on the formula, instead

⋆ Supported by the UK EPSRC project *Complete and Efficient Checks for Branching-Time Abstractions* (EP/E028985/1).

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 89–104, 2009.

of EXPTIME-complete and quadratic as previously stated erroneously in [BG00]. The definition of GMC depends on the exact notion of abstraction, and is usually tailored for branching-time model checking [BG00]. We then study a simpler *linear completeness preorder* for relating program abstractions. We show that LTL GMC with this weaker preorder is only EXPSPACE-complete in the size of the formula, and can be solved in linear time and logarithmic space in the size of the model. Finally, we identify classes of formulas for which the model complexity of GMC defined with the standard branching-time completeness preorder is reduced.

Example. Consider the program P:

```
program P() {
  x,y = 1,0;
  x,y = 2*f(x),f(y);
  x,y = 1,0;
}
```

where `x` and `y` denote `int` variables, `f : int -> int` denotes some unknown function, and the notation "`x,y = 1,0`" means variables `x` and `y` are simultaneously assigned values `1` and `0`, respectively. Let ϕ_1 denote the LTL formula $Fq_y \wedge G(q_x \vee \neg q_y)$ with the two predicates q_x : "is x odd?" and q_y : "is y odd?", and where F means "eventually" while G means "always", and let ϕ_2 denote the LTL formula $Xq_y \wedge G(q_x \vee \neg q_y)$, where X means"next" (see the next section for formal definitions).

Given such a program and knowing the predicate of interests q_x and q_y, predicate abstraction can be used to automatically generate the following 3-valued Kripke structure M (or "Boolean program" [BR01]) abstracting P [GHJ01]:

initial state s_0: $q_x = \mathit{true},\ q_y = \mathit{false}$
next state s_1: $q_x = \mathit{false}, q_y = \bot$
next state s_2: $q_x = \mathit{true},\ q_y = \mathit{false}$
loop forever in s_2

As shown in [GJ02] and discussed later, model checking[1] ϕ_1 and ϕ_2 against M returns the value "unknown," while generalized model checking can prove that no concretization of M can possibly satisfy either ϕ_1 or ϕ_2, i.e., no matter how function `f` is implemented.

Although $\phi_2 = Xq_y \wedge G(q_x \vee \neg q_y)$ is an LTL safety formula and hence is within the scope of predicate-abstraction-based software model checkers such as SLAM [BR01] or BLAST [HJMS02], these tools cannot prove that ϕ_2 does not hold regardless of the definition of function `f`: this result can only be obtained through generalized model checking. Instead, when confronted with such a program P, these tools would attempt to iteratively refine the abstraction M by analyzing the code of function `f` if it is available. This process is in general exponential in the size of the abstraction, since adding a single predicate in each iteration may double the size of the abstraction. Moreover, this process may not terminate. For the above abstraction M and formula ϕ_2, the expensive and unpredictable abstraction-refinement process can thus be avoided thanks to GMC. Although the worst-case complexity of GMC is expensive in the size of the (usually

[1] In *model checking*, we mean normal 3-valued model checking in the sense of [BG99].

short) formula (but so is traditional LTL model checking which is already PSPACE-complete), GMC can always be done in time polynomial in the size of the model (linear or quadratic in many cases as shown later), in contrast with abstraction refinement which is typically exponential in the (usually large) model. □

2 Preliminaries

A *partial Kripke structure* (PKS for short) [BG99] is $M = \langle S, R, L, s^{in}\rangle$ where S is a nonempty set of states, $R \subseteq S \times S$ is a total image-finite transition relation (i.e., every state has a non-zero finite number of immediate successor states), $L : S \times AP \to \mathbf{3}$ is a labeling of states that associates a truth value in $\mathbf{3} = \{true, \bot, false\}$ to each atomic proposition in a finite set AP, and $s^{in} \in S$ is an initial state. For a state s and proposition p, we say that p is true in s if $L(s,p) = true$, it is false in s if $L(s,p) = false$, and it is unknown $\bot$ otherwise. A PKS is *complete* if the range of L is $\mathbf{2} = \{true, false\}$. We call a complete PKS a *Kripke Structure* or KS. When we want to stress that a PKS M is complete, we denote it by $\overline{M}$. Given a state s, we denote by $L(s)$ the function $\sigma : AP \to \mathbf{3}$ such that $\sigma(p) = L(s,p)$. We use the notations $\mathbf{3}^{AP} = \{\sigma : AP \to \mathbf{3}\}$ and $\mathbf{2}^{AP} = \{\sigma : AP \to \mathbf{2}\}$. For $s \in S$, we denote by (M,s) the PKS $\langle S, R, L, s\rangle$.

A *computation* of M is $s_0, s_1, \ldots$ such that $s_0 = s^{in}$ and forall $i \geq 0$ we have $(s_i, s_{i+1}) \in R$. A computation $\pi = s_0, s_1, \ldots$ induces a *trace* $L(\pi) = L(s_0)L(s_1)\cdots \in (\mathbf{3}^{AP})^{\omega}$. The set of computations of M is denoted $\mathcal{C}(M)$ and the set of traces of M is denoted $\mathcal{L}(M)$. In general, $\mathcal{L}(M) \subseteq (\mathbf{3}^{AP})^{\omega}$. Given a PKS $M = \langle S, R, L, s^{in}\rangle$, the *unwinding* of M into a tree is the PKS $M^+ = \langle S^+, R', L', s^{in}\rangle$, where S^+ is the set of nonempty sequences over S, $R' = \{(s_1 \cdots s_n, s_1 \cdots s_n \cdot s_{n+1}) \in (S^+ \times S^+) \mid (s_n, s_{n+1}) \in R\}$, and $L'(\pi \cdot s) = L(s)$. We restrict the set S^+ to the set of sequences reachable from s^{in}. If M is a Kripke structure then so is M^+.

To interpret temporal logic formulas on PKSs, we extend Kleene's strong 3-valued propositional logic [Kle87]. Conjunction $\wedge$ in this logic is defined as the minimum Min of its arguments with respect to the *truth ordering* $\leq_T$ where $false \leq_T \bot \leq_T true$. We extend this function to sets in the obvious way, with $Min(\emptyset) = true$. Negation $\neg$ is defined using the function '$Comp$' that maps $true$ to $false$, $false$ to $true$, and $\bot$ to $\bot$. Disjunction $\vee$ is defined as usual using De Morgan's laws: $p \vee q = \neg(\neg p \wedge \neg q)$. Propositional modal logic (PML) is propositional logic extended with the modal operator AX (which is read "for all immediate successors"). Formulas of PML have the following abstract syntax: $\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid AX\phi$, where p ranges over AP. The following 3-valued semantics generalizes the traditional 2-valued semantics for PML.

Definition 1. *The value of a formula ϕ of 3-valued PML in a state s of a PKS $M = \langle S, R, L, s^{in}\rangle$, written $[(M,s) \models \phi]$, is defined inductively as follows:*

$$
\begin{aligned}
[(M,s) \models p] &= L(s,p)\\
[(M,s) \models \neg\phi] &= Comp([(M,s) \models \phi])\\
[(M,s) \models \phi_1 \wedge \phi_2] &= Min(\{[(M,s) \models \phi_1], [(M,s) \models \phi_2]\})\\
[(M,s) \models AX\phi] &= Min(\{[(M,s') \models \phi] \mid (s,s') \in R\})
\end{aligned}
$$

We write $[M \models \phi]$ for $[(M, s^{in}) \models \phi]$. This 3-valued logic can be used to define a preorder $\preceq$ on PKSs that reflects their degree of completeness. Let $\leq_I$ be the *information ordering* on truth values where $\bot$ is the least element and $true$ and $false$ are maximal uncomparable elements: $\bot \leq_I true, false$. For two PKS $M_i = \langle S_i, R_i, L_i, s_i^{in} \rangle$ with $i = 1, 2$ the *completeness preorder* is the greatest relation $\preceq \subseteq S_1 \times S_2$ such that $s_1 \preceq s_2$ implies all the following:

1. For every $p \in AP$, we have $L_1(s_1, p) \leq_I L_2(s_2, p)$.
2. For every $(s_1, s_1') \in R_1$, there exists $(s_2, s_2') \in R_2$ such that $s_1' \preceq s_2'$.
3. For every $(s_2, s_2') \in R_2$, there exists $(s_1, s_1') \in R_1$ such that $s_1' \preceq s_2'$.

We say that M_2 is *more complete* than M_1, denoted $M_1 \preceq M_2$, if $s_1^{in} \preceq s_2^{in}$. It can be shown that 3-valued PML logically characterizes the completeness preorder.

Theorem 1. [BG99] *Let M_1 and M_2 be partial Kripke structures, and let Φ be the set of all formulas of 3-valued PML. Then $M_1 \preceq M_2$ iff $(\forall \phi \in \Phi : [M_1 \models \phi] \leq_I [M_2 \models \phi])$.*

In other words, partial Kripke structures that are "more complete" with respect to $\preceq$ have more definite properties with respect to $\leq_I$, i.e., have more properties that can be established $true$ or $false$ by model checking. Moreover, any formula ϕ of 3-valued PML that evaluates to $true$ or $false$ on a partial Kripke structure has the same truth value when evaluated on any more complete structure.

2.1 Model Checking and Generalized Model Checking

The sets of LTL and CTL formulas are defined as follows.

$$\text{LTL } \varphi ::= p \mid \varphi \wedge \varphi \mid \neg\varphi \mid X\varphi \mid \varphi U \varphi$$

$$\text{CTL } \varphi ::= p \mid \varphi \wedge \varphi \mid \neg\varphi \mid AX\varphi \mid A\varphi U \varphi \mid E\varphi U \varphi$$

We assume familiarity with the semantics of LTL and CTL and with their model checking. As usual, we denote $true\, U\varphi$ by $F\varphi$, $\neg F \neg \varphi$ by $G\varphi$ and $\neg((\neg\psi)U(\neg\varphi \wedge \neg\psi))$ by $\varphi R \psi$. The above grammar includes a complete set of operators and other operators can be expressed in the usual way. Given a set of propositions AP and an LTL formula φ, the language of φ, denoted $L(\varphi)$ is the set of models of φ in $(\mathbf{2}^{AP})^\omega$. Formally, $L(\varphi) = \{w \in (\mathbf{2}^{AP})^\omega \mid w \models \varphi\}$. The 3-valued semantics of LTL and CTL path formulas extend Definition 1 as expected. For instance, given a 3-valued infinite word $w = a_0 a_1 a_2 \cdots \in (\mathbf{3}^{AP})^\omega$, $[w \models X\varphi] = [w' \models \varphi]$ with $w' = a_1 a_2 \cdots \in (\mathbf{3}^{AP})^\omega$, while $[w \models \varphi_1 U \varphi_2] = Max(\{Min(\{[a_i \models \varphi_1] | i < k\} \cup \{[a_k \models \varphi_2]\}) | k \geq 0\})$. For partial Kripke structure M and a CTL formula ϕ, we denote the value of ϕ at state s by $[(M, s) \models \phi] \in \mathbf{3}^{AP}$. For the initial state s^{in} of M we denote $[(M, s^{in}) \models \phi]$ by $[M \models \phi]$. If M is a Kripke structure we simply write $M, s \models \varphi$ for $[(M, s) \models \varphi] = true$ and $M, s \not\models \varphi$ for $[(M, s) \models \varphi] = false$. For a Kripke structure $\overline{M}$ and an LTL formula φ, we say that $\overline{M}$ satisfies φ, denoted $\overline{M} \models \varphi$ if $\mathcal{L}(\overline{M}) \subseteq L(\varphi)$.

In practice, the size of the Kripke structure $\overline{M}$ can be prohibitively expensive or even infinite. Instead, a smaller (finite) *abstraction* M' can be used: if M' is generated in such a way that $M' \preceq \overline{M}$, then all the properties ϕ that can be proved ($true$) or disproved ($false$) on M' will also hold on $\overline{M}$, by Theorem 1. With static program analysis

and predicate abstraction, generating such abstractions with respect to the completeness preorder $\preceq$ can be done at the same computational cost as computing standard abstractions that merely simulate (over-approximate) the concrete system $\overline{M}$ [GHJ01]. Moreover, 3-valued model checking can itself be done at the same computational cost as regular 2-valued model checking [BG00].

In some cases, precisely characterized in [GH05] and also independently studied in [GC05], all the completions of an abstraction M agree on the satisfaction of a formula φ, yet 3-valued model checking is not accurate enough to identify this and still returns $\bot$. For instance, this is the case for the formula $p \vee \neg p$ if p is $\bot$. This observation suggests a more precise version of 3-valued model checking [BG00]: the value of a formula φ in a PKS M should be unknown only if some completions of M satisfy φ *and* some completions of M falsify φ [BG00]. We denote the value of φ on M according to this *thorough semantics* by $[M \models \varphi]_t \in \mathbf{3}$.

Generalized model checking (GMC) can determine the value of $[M \models \varphi]_t$ [BG00]. Given a PKS M and a formula φ, the GMC problem for M and φ is to determine whether there exists a Kripke structure M' that completes M and satisfies φ. Formally, we have the following.

$$M \models_{\preceq} \varphi \text{ iff there exists } \overline{M'} \succeq M \text{ such that } \overline{M'} \models \varphi$$

The value $[M \models \varphi]_t$ can be evaluated with two GMC questions. First, we check whether $M \models_{\preceq} \varphi$. If the answer is no, then all completions of M do not satisfy φ and $[M \models \varphi]_t = \mathit{false}$. If the answer is yes, we next check whether $M \models_{\preceq} \neg\varphi$. If that answer is no, then we know that all completions of M satisfy φ and $[M \models \varphi]_t = \mathit{true}$. Otherwise, $[M \models \varphi]_t = \bot$.

It can be shown that 3-valued model checking is sound with respect to the thorough semantics.

Theorem 2. [BG00] *Let M be a PKS and φ an LTL or CTL formula.*

1. $[M \models \varphi] = \mathit{true}$ *implies* $[M \models \varphi]_t = \mathit{true}$.
2. $[M \models \varphi] = \mathit{false}$ *implies* $[M \models \varphi]_t = \mathit{false}$.

In this paper we revisit LTL generalized model checking and show that its complexity is greater than what was previously believed. We also consider specifications (both in LTL and CTL) for which the model complexity of generalized model checking is simpler than the general case.

2.2 Automata over Infinite Words

We assume familiarity with the basic notions of alternating automata on infinite words, cf. [GTW02]. We also refer to tree automata, however, we do not define them formally.

For an alphabet Σ, the set Σ^* is the set of finite sequences of elements from Σ. For $x \in \Sigma^*$, we denote the length of x by $|x|$. Given an alphabet Σ and a set D of directions, a *Σ-labeled D-tree* is a pair $\langle T, \tau\rangle$, where $T \subseteq D^*$ is a tree over D and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ.

For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas true and false. An alternating word automaton is $A = \langle \Sigma, Q, q_{in}, \delta, \alpha\rangle$, where

Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is a transition function, $q_{in} \in Q$ is an initial state, and α specifies the acceptance condition. A run of A on $w = \sigma_0\sigma_1 \cdots$ is a Q-labeled D-tree, $\langle T, \tau\rangle$, where $\tau(\epsilon) = q_{in}$ and, for every $x \in T$, we have $\{\tau(x \cdot \gamma_1), \ldots, \tau(x \cdot \gamma_k)\} \models \delta(\tau(x), \sigma_{|x|})$ where $\{x \cdot \gamma_1, \ldots, x \cdot \gamma_k\}$ is the set of children of x. A run of A is accepting if all its infinite paths satisfy the acceptance condition. For a path π, we denote the set of automaton states visited infinitely often along this path by $inf(\pi)$. We consider the following three acceptance conditions:

- A path π satisfies a *Büchi* condition $\alpha \subseteq Q$ iff $inf(\pi) \cap \alpha \neq \emptyset$.
- A path π satisfies a *co-Büchi* condition $\alpha \subseteq Q$ iff $inf(\pi) \cap \alpha = \emptyset$.
- A path π satisfies a *parity* condition $\alpha = \langle F_0, \ldots, F_k\rangle$ where $F_0, \ldots F_k$ form a partition of Q iff for some even i we have $inf(\pi) \cap F_i \neq \emptyset$ and forall $i' < i$ we have $inf(\pi) \cap F_{i'} = \emptyset$. We call k the number of *priorities* of α.

For the three conditions, an automaton accepts a word iff there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-words that A accepts.

Below we discuss some special cases of alternating automata. The alternating automaton A is *nondeterministic* if for all the formulas that appear in δ are disjunctions over the states Q. The automaton A is *deterministic* if all formulas that appear in δ are states from Q. For a nondeterministic automaton we write $\delta : Q \times \Sigma \to 2^Q$ and for a deterministic automaton we write $\delta : Q \times \Sigma \to Q$.

We denote each of the different types of automata by an acronym in $\{D, N, A\} \times \{W, B, C, P\} \times \{W, T\}$, where the first letter describes the branching mode of the automaton (deterministic, nondeterministic, or alternating), the second letter describes the acceptance condition (Weak,[2] Büchi, co-Büchi, or parity), and the third letter describes the object over which the automaton runs (words or trees). For example, an ABW is an alternating Büchi word automata and a DPW is a deterministic parity word automata.

We state the following well known results about automata and their relation to LTL.

Theorem 3. *For every LTL formula φ of length n there exist an NBW N_φ with $2^{O(n)}$ states* [VW94] *and a DPW D_φ with $2^{2^{O(n\ log n)}}$ states and $2^{O(n)}$ priorities* [Saf88, Pit07] *such that $L(\varphi) = L(N_\varphi) = L(D_\varphi)$.*

Theorem 4. [Jur00] *Given an APW A over a 1-letter alphabet with n states and k priorities, we can decide whether $L(A) = \emptyset$ in time proportional to $n^{O(k)}$.*

Theorem 5. [SVW87] *Given two NBW N_1, N_2 we can decide whether $L(N_1) \subseteq L(N_2)$ in space logarithmic in N_1 and polynomial in N_2.*

3 LTL Generalized Model Checking

We show that, contrary to previous beliefs, GMC with respect to linear time logic is 2EXPTIME-complete. Our upper bound combines a DPW for the LTL property with the PKS to get an APW over a 1-letter alphabet. The APW is not empty iff the GMC problem holds. For the lower bound, we show a reduction from LTL realizability to

[2] We delay the definition of weak automata to Section 5.

generalized model checking. LTL realizability is 2EXPTIME-hard [PR89] establishing 2EXPTIME-hardness of generalized model checking. The two together establish 2EXPTIME-completeness of generalized model checking for LTL.

Theorem 6. *LTL generalized model checking $M \models_{\preceq} \varphi$ can be solved in polynomial time in the size of M and double exponential time in the size of φ.*

Proof. Consider an LTL formula φ. Let $|\varphi| = n$. According to Theorem 3 there exists a DPW D_φ with $2^{2^{O(n \log n)}}$ states and $2^{O(n)}$ priorities such that $L(\varphi) = L(D_\varphi)$.

Let $D_\varphi = \langle \mathbf{2}^{AP}, T, t_0, \rho, \alpha \rangle$ and $M = \langle S, R, L, s^{in} \rangle$. Consider the following APW A over a 1-letter alphabet that is obtained from the combination of M and D_φ. We define $A = \langle \{a\}, T \times S, (t_0, s^{in}), \eta, \alpha' \rangle$ such that

$$\eta((t,s),a) = \bigvee_{\overline{\sigma} \succeq L(s)} \bigwedge_{(s,s') \in R} (\rho(t, \overline{\sigma}), s')$$

and $\alpha' = \langle F'_0, \ldots, F'_k \rangle$ is obtained from $\alpha = \langle F_0, \ldots, F_k \rangle$ by setting $F'_j = F_j \times S$.

Lemma 1. *A accepts a^ω iff $M \models_{\preceq} \varphi$.*

According to Theorem 4 the emptiness of A can be determined in time proportional to $(2^{2^{O(n \log n)}})^{2^{O(n)}} = 2^{2^{O(n \log n)}}$. □

Note that, if D_φ was nondeterministic in the previous proof, it could not precisely track simultaneously different matching states s such that $s \preceq s_n$ in the proof, and therefore $M \models_{\preceq} \varphi$ would not necessarily imply that A accepts a^ω. This is in essence the error in the proof of Theorem 25 of [BG00], which led to the overly optimistic EXPTIME upper-bound.

We now proceed to the lower bound. We start with a definition of LTL realizability. Consider a set of propositions $AP = I \cup O$ of input and output signals, respectively. Let L be a language of infinite words over alphabet $\mathbf{2}^{AP}$. The *realizability problem* for L is to decide whether there exists a strategy $f : (2^I)^+ \rightarrow 2^O$ such that all the computations generated by f are in L. A *computation* $\pi = (i_0, o_0), (i_1, o_1), \ldots$ is generated by f if for all $j \geq 0$ we have $o_j = f(i_0 i_1 \cdots i_j)$. The realizability problem for an LTL formula φ is the realizability problem for $L(\varphi)$.

Theorem 7. [PR89] *The realizability problem for an LTL formula φ is 2EXPTIME-hard in the size of φ.*

Theorem 8. *LTL Generalized model checking $M \models_{\preceq} \varphi$ is 2EXPTIME-hard in the size of φ.*

Proof. We show how to solve realizability of an LTL formula using the generalized model checking problem. The idea behind the reduction is that the PKS includes determined values of the inputs and undetermined values of the outputs. The branching of the PKS forces all possible assignments to inputs as possible successors of every state. Thus, every completion of the PKS associates an assignment to the outputs with every possible assignment to inputs and is in essence a strategy. If the completion satisfies the

LTL formula, then so does the strategy. The PKS has 2^I different states, each labeled by the appropriate assignment to the input variables and with transitions between every two possible states. We then show how to reduce the PKS to one with a constant number of states and $|O| + 2$ propositions. □

4 Linear Completeness Preorder

The completeness preorder $\preceq$ used to define generalized model checking $\models_{\preceq}$ is stronger than necessary for reasoning only about the linear behaviors of partial Kripke structures. Indeed, the completeness preorder reduces to a bisimulation relation in the case of complete Kripke structures, and Kripke structures that satisfy the same LTL formulas are not necessarily bisimilar.

In this section, we study a simpler *linear completeness preorder* $\preceq_L$, first suggested in [BG00], that relates partial Kripke structures using only their sets of (3-valued) traces. Then we show that generalized model checking $\models_{\preceq_L}$ defined with respect to this linear preorder is "only" EXPSPACE-complete.

Given any two infinite 3-valued traces $w{=}L(s_0)L(s_1)\cdots$ and $w'{=}L(s'_0)L(s'_1)\cdots$ in $(\mathbf{3}^{AP})^\omega$, we write $w \leq_I w'$ if $\forall i \geq 0 : \forall p \in AP : L(s_i, p) \leq_I L(s'_i, p)$.

Definition 2. *For two PKS $M_i = \langle S_i, R_i, L_i, s_i^{in}\rangle$ with $i = 1, 2$, the* linear completeness preorder $\preceq_L$ *is the greatest relation $\preceq_L \subseteq S_1 \times S_2$ such that $(s_1, s_2) \in \preceq_L$ implies all the following.*

1. *For every $w \in \mathcal{L}(M_1, s_1)$ there exists $w' \in \mathcal{L}(M_2, s_2)$ such that $w \leq_I w'$.*
2. *For every $w' \in \mathcal{L}(M_2, s_2)$ there exists $w \in \mathcal{L}(M_1, s_1)$ such that $w \leq_I w'$.*

It is easy to show that 3-valued LTL logically characterizes the linear completeness preorder.

Theorem 9. *For any two PKS M_1 and M_2, we have $M_1 \preceq_L M_2$ iff for every LTL formula φ we have $[M_1 \models \varphi] \leq_I [M_2 \models \varphi]$.*

Proof. Assume $M_1 \preceq_L M_2$ and consider any LTL formula φ. If $[M_1 \models \varphi] = \bot$, we always have $[M_1 \models \varphi] \leq_I [M_2 \models \varphi]$.

If $[M_1 \models \varphi] = \mathit{true}$, then for all $w \in \mathcal{L}(M_1)$, $[w \models \varphi] = \mathit{true}$. By point 2 of Definition 2, for every $w' \in \mathcal{L}(M_2)$ there exists $w \in \mathcal{L}(M_1)$ such that $w \leq_I w'$. But since $\forall w \in \mathcal{L}(M_1) : [w \models \varphi] = \mathit{true}$, we have $\forall w' \in \mathcal{L}(M_2) : [w' \models \varphi] = \mathit{true}$, and hence $[M_2 \models \varphi] = \mathit{true}$.

If $[M_1 \models \varphi] = \mathit{false}$, then $\exists w \in \mathcal{L}(M_1) : [w \models \varphi] = \mathit{false}$. By point 1 of Definition 2, we have $\exists w' \in \mathcal{L}(M_2) : w \leq_I w'$ and hence $[w' \models \varphi] = \mathit{false}$. Thus $[M_2 \models \varphi] = \mathit{false}$, and the first direction of the theorem holds.

Conversely, let $s_1 \sqsubseteq s_2$ denote $\forall \varphi \in LTL : [(M_1, s_1) \models \varphi] \leq_I [(M_2, s_2) \models \varphi]$. Assume that $s_1 \sqsubseteq s_2$ but that $s_1 \not\preceq_L s_2$: thus, either point 1 or 2 of Definition 2 is violated.

Assume point 1 is violated: $\exists w \in \mathcal{L}(M_1, s_1) : \forall w' \in \mathcal{L}(M_2, s_2) : w \not\leq_I w'$. Let $w = s_1^0 s_1^1 s_1^2 \cdots$ with $s_1^0 = s_1$. Let $S_2^0 = \{s_2\}$ and for $k > 0$, let $S_2^k = \{s \in S_2 \mid s' \in S_2^{k-1} \wedge (s', s) \in R_2 \wedge (\forall p \in AP : L_1(s_1^k, p) \leq_I L_2(s, p))\}$. Since $\forall w' \in \mathcal{L}(M_2, s_2) : w \not\leq_I w'$, then there must exist a value of k such that $S_2^k = \emptyset$. In other words, the

corresponding s_1^k in M_1 denote the first state in M_1 reachable from s_1 along w whose label cannot be "matched" (according to the previous formal definition) by any state of M_2 (hence also reachable in k steps from s_2). By abusing notation, let $S_2^k = \{s \in S_2 \mid s' \in S_2^{k-1} \wedge (s', s) \in R_2\}$ (by construction, we know $S_2^{k-1} \neq \emptyset$ and since every state has at least one successor state, S_2^k is nonempty as well). Thus, for each state $s \in S_2^k$, there exists a proposition $p \in AP$ such that $L_1(s_1^k, p) \not\leq_I L_2(s, p)$. Let $\varphi(s) = p$ if $L_1(s_1^k, p) = \mathit{false}$ and let $\varphi(s) = \neg p$ otherwise (i.e., when $L_1(s_1^k, p) = \mathit{true}$; if $L_1(s_1^k, p) = \bot$, then trivially $L_1(s_1^k, p) \leq_I L_2(s, p)$). Consider the LTL formula

$$\psi = (\bigwedge_{i<k}(X^i(\bigwedge_{L(s_1^i,p)=true} p \wedge \bigwedge_{L(s_1^i,p)=false} \neg p))) \Rightarrow X^k \bigvee_{s \in S_2^k} \varphi(s)$$

We have $[(M_1, s_1) \models \psi] = \mathit{false}$ (as we know $[w \models \psi] = \mathit{false}$) while $[(M_2, s_2) \models \psi] \neq \mathit{false}$ (since the antecedent of the logical implication is true exactly for finite paths leading to states in S_2^{k-1} and the consequent is either true or $\bot$ for all states in S_2^k). A contradiction with $s_1 \sqsubseteq s_2$.

Assume point 2 is violated: $\exists w' \in \mathcal{L}(M_2, s_2) : \forall w \in \mathcal{L}(M_1, s_1) : w \not\leq_I w'$. Using the same line of reasoning as in the previous case, let s_2^k denote the first state in M_2 reachable from s_2 along w' whose label cannot be matched by any state in S_1^k of M_1 as defined above. Thus, for each state $s \in S_1^k$, there exists a proposition $p \in AP$ such that $L_1(s, p) \not\leq_I L_2(s_2^k, p)$. Let $\varphi(s) = p$ if $L_1(s, p) = \mathit{true}$ and let $\varphi(s) = \neg p$ otherwise. Consider the LTL formula

$$\psi = (\bigwedge_{i<k}(X^i(\bigwedge_{L(s_2^i,p)=true} p \wedge \bigwedge_{L(s_2^i,p)=false} \neg p \wedge \bigwedge_{L(s_2^i,p)=\bot} (p \wedge \neg p)))) \Rightarrow X^k \bigvee_{s \in S_1^k} \varphi(s)$$

We have $[(M_1, s_1) \models \psi] = \mathit{true}$ (since the antecedent of the logical implication is either true or $\bot$ exactly for the finite paths leading to states in S_1^{k-1} and the consequent is true for all states in S_1^k) while $[(M_2, s_2) \models \psi] \neq \mathit{true}$ (since $[w' \models \psi] \neq \mathit{true}$). A contradiction with $s_1 \sqsubseteq s_2$. □

Given a PKS M and an LTL formula φ, generalized model checking with respect to the linear completeness preorder $\preceq_L$ means checking whether every 3-valued trace of M can be completed to a 2-valued trace that satisfies φ. Formally, we have the following.

$$M \models_{\preceq_L} \varphi \text{ iff } \forall w \in \mathcal{L}(M) : \exists \text{ a complete } w' \text{ such that } w \leq_I w' \text{ and } w' \models \varphi$$

As observed in [GJ02], computing the value of $[M \models \varphi]_t$ for an LTL formula φ can be reduced to one normal (2-valued) model checking problem and one generalized model checking problem, regardless of which completeness preorder is used. One can start by checking whether there exists a completion w' of any trace w in M such that $w' \not\models \varphi$. To do this, one can build a Kripke structure M^c that guesses all possible completions of labelings of states of M and thus accepts all the possible completions of traces of M. Then, one checks whether $M^c \models \varphi$ using traditional 2-valued LTL model checking, which is a PSPACE-complete problem. If $M^c \models \varphi$, all possible completions of M satisfy φ, which means $[M \models \varphi]_t = \mathit{true}$ and we stop. Otherwise, one needs

to solve a second, more expensive generalized model checking problem to determine whether there exists some completion M' of M whose traces all satisfy φ.

If one considers the completeness preorder $\preceq$, checking for such a completion $M' \succeq M$ such that $M' \models \varphi$, i.e., computing $M \models_{\preceq} \varphi$, is 2EXPTIME-complete as shown in the previous section. However, if one considers instead the *linear* completeness preorder $\preceq_L$, we now show that computing $M \models_{\preceq_L} \varphi$ is only EXPSPACE-complete.

Theorem 10. *LTL generalized model checking $M \models_{\preceq_L} \varphi$ with respect to the linear completeness preorder $\preceq_L$ can be solved in space logarithmic in the size of M and exponential in the size of φ.*

Proof. Consider an LTL formula φ. According to Theorem 3 there exists an NBW $N_\varphi = \langle \mathbf{2}^{AP}, Q, q_0, \rho, F\rangle$ where $|Q| = 2^{O(|\varphi|)}$ such that $L(N_\varphi) = L(\varphi)$.

We modify the NBW above to an NBW over the alphabet $\mathbf{3}^{AP}$ that accepts partial traces that have a completion in $L(N_\varphi)$. Formally, we have the following.

We denote letters in $\mathbf{2}^{AP}$ by $\overline{\sigma}$ and letters in $\mathbf{3}^{AP}$ by τ. Let N' be the automaton obtained from N_φ by guessing a completion of the read letter. Formally, $N' = \langle \mathbf{3}^{AP}, Q, q_0, \rho', F\rangle$ where

$$\rho'(s,\tau) = \bigvee_{\overline{\sigma} \succeq \tau} \rho(s, \overline{\sigma})$$

Now, all that we have to check is whether $L(M) \subseteq L(N')$. From Theorem 5, we know that this problem can be solved in space logarithmic in M and polynomial in N'. As N' is exponential in φ, the upper bound follows. □

We now show that using this definition of GMC we can solve an EXPSPACE-hard tiling problem [vEB97]. In tiling problems we get a finite set of different types of tiles and we have to tile a floor of a given dimension. We may use as many tiles as we want from every given type, however, there are rules that tell us which tiles are allowed to be next to each other according to vertical and horizontal rules. There are many different flavors of tiling problems with different complexities. Here we introduce the EXPSPACE version of the tiling problem. In order to prove the lower bound, we build a PKS M whose traces are all the possible arrangements of tiles. A trace has a completion that satisfies our LTL formula φ if the arrangement of tiles is not valid, i.e., it violates one of the tiling rules. That is, $M \models_{\preceq_L} \varphi$ iff all possible arrangements of tiles are not valid, i.e., the tiling problem does not have a solution.

A *tiling problem* is $\langle T, H, V, s, t, n\rangle$, where T is a finite set of tiles, $H, V \subseteq T \times T$ are horizontal and vertical consistency rules, $s, t \in T$ are initial and final tiles, and n is a number (in unary). The decision problem is whether there exists a number m and a function $f : [2^n] \times [m] \to T$ such that $f(1,1) = s$, $f(2^n, m) = t$, and forall i, j we have $(f(i,j), f(i+1,j)) \in H$ and $(f(i,j), f(i,j+1)) \in V$. That is, arrange the tiles in a 2^n times m rectangle such that s is in the bottom left corner, t in the top right corner, and all neighbors (vertical/horizontal) satisfy the horizontal and vertical consistency rules. This problem is EXPSPACE-complete [vEB97].

Theorem 11. *LTL generalized model checking $M \models_{\preceq_L}$ with respect to the linear completeness preorder $\preceq_L$ is EXPSPACE-hard in the size of φ.*

Proof. We start by representing the rectangular arrangement of tiles by a linear sequence of tiles. An (infinite) linear sequence of tiles represents a valid tiling if it starts with s, has t in location $m2^n$ for some m, every adjacent locations (except multiples of 2^n and their successors) satisfy H, and every two locations whose distance is 2^n satisfy V.

We construct a simple system that produces all possible sequences of tiles. The partial propositions are going to number every tile in the sequence with a number in $[0..(2^n - 1)]$. The LTL formula checks two things. First, that the truth assignments to partial propositional variables behave like a counter (it is always possible to complete the values of these propositions in this way). Second, that every possible sequence of tiles contains one of the following problems: either (a) it does not start in s, or (b) all locations that are multiples of 2^n are not t, or (c) the horizontal rule is violated before t appears in a 2^n-multiple location, or (d) the vertical rule is violated before t appears in a 2^n-multiple location. If one of these problems occurs, then the tiling is not valid. If all possible arrangements of tiles are not valid, then the tiling problem does not have a solution. As before, we show also how to reduce the structure to one with a constant number of states. □

The next theorem states that $\preceq$ is a stronger relation than $\preceq_L$, which in turn helps explain why checking $\models_{\preceq}$ is more expensive than checking $\models_{\preceq_L}$.

Theorem 12. *For any partial Kripke structures M, M' and LTL formula φ, $M \preceq M'$ implies $M \preceq_L M'$, and therefore $M \models_{\preceq} \varphi$ implies $M \models_{\preceq_L} \varphi$.*

Proof. Immediate from the definitions of $\preceq$ and $\preceq_L$. □

Note that $\preceq$ is *strictly* stronger than $\preceq_L$, as the converse of the theorem does not hold. To illustrate this, consider the LTL formula $\varphi = (p \wedge Xp) \vee (\neg p \wedge X\neg p)$ and the partial Kripke structure $M = \langle\{s_0, s_1, s_2\}, \{(s_0, s_1), (s_0, s_2), (s_1, s_1), (s_2, s_2)\}, L, s_0\rangle$ labeled with a single atomic proposition p such that $L(s_0, p) = \bot$, $L(s_1, p) = \mathit{true}$ and $L(s_2, p) = \mathit{false}$. It is easy to see that $[(M, s_0) \models \varphi] = \bot$. Moreover, we have $(M, s_0) \models_{\preceq_L} \varphi$, as every 3-valued trace generated from (M, s_0) can be completed by some 2-valued trace that satisfies φ. However, $(M, s_0) \not\models_{\preceq} \varphi$ as there does not exist a completion M' such that $M \preceq M'$ and $M' \models \varphi$, as state s_0 where $p = \bot$ cannot be completed to a *single* state s such that every trace from s satisfies φ: if $L(s, p) = \mathit{true}$, then the trace ss_2^ω violates φ, and if $L(s, p) = \mathit{false}$, then the trace ss_1^ω violates φ.

5 Model Complexity

We have seen that LTL generalized model checking defined with the stronger branching-time preorder $\preceq$ is polynomial in the size of the model. The degree of the polynomial, however, is unbounded, and depends on the deterministic automaton created for the formula. Here we show that for interesting classes of properties, the model complexity can be restricted to linear or quadratic. The resemblance pointed out between generalized model checking and realizability in the proof of Theorem 8 continues here. Indeed, the same classes of formulas are used to suggest tractable fractions of LTL for realizability (cf. [RW89, AMPS98, PPS06]).

We start with a few additional definitions and known results regarding automata. Let $\mathcal{A} = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$ be a Büchi automaton. We say that $\mathcal{A}$ is *weak* if there is a preorder $\leq$ on the state set Q such that the following two conditions hold:

1. For every $q \in Q$ and $\sigma \in \Sigma$, if q' appears in $\delta(q, \sigma)$ then $q \leq q'$.
2. For every $q \in Q$, if $q \in \alpha$ then forall q' such that $q \leq q'$ and $q' \leq q$ we have $q' \in \alpha$.

We use the acronyms mentioned previously for weak automata. For instance, an AWT is an alternating weak tree automaton and an DWW is a deterministic weak word automaton.

We specialize Theorem 4 to our needs as follows.

Theorem 13. *Given an APW A over a 1-letter alphabet, we can decide whether $L(A) = \emptyset$ in linear time if A is AWW [KVW00] and in quadratic time if A is an ABW, ACW, or an APW with three priorities [VW86, Jur00].*

Consider an LTL formula φ. We say that φ is a *safety property* if for every word $w \notin L(\varphi)$ there exists a prefix u such that forall v' we have $uv' \notin L(\varphi)$. Let p and q be Boolean combinations of propositional formulas. Formulas of the form GFp or $G(q \rightarrow Fp)$ are called *response properties*, and formulas of the form FGp are called *persistence properties* [MP92]. If φ is of the form $(\varphi_s^a \wedge \varphi_r^a) \rightarrow (\varphi_s^g \wedge \varphi_r^g)$ where φ_s^a and φ_s^g are conjunctions of safety properties and φ_r^a and φ_r^g are conjunctions of response properties is called *generalized reactivity[1]* [KPP03]. Alternatively, we classify LTL properties according to the type of deterministic automaton that accepts the same language. We say that φ is a *weak property* if there exists a DWW that accepts the language of φ. We say that φ is a *DBW property* if there exists a DBW that accepts the language of φ. Similarly, we say that φ is a *DCW property* if there exists a DCW that accepts the language of φ. The following theorem links the different types of LTL properties to the deterministic automata that accept them.

Theorem 14

1. *For every safety property φ, there exists a DWW D such that $L(D) = L(\varphi)$.*
2. *For every response property φ, there exists a DBW D such that $L(D) = L(\varphi)$.*
3. *For every persistence property φ, there exists a DCW D such that $L(D) = L(\varphi)$.*
4. *For every generalized reactivity[1] property φ, there exists a DPW D with three priorities such that $L(D) = L(\varphi)$.*

The following is a consequence of Theorems 13 and 14 and the proof of Theorem 6.

Theorem 15. *LTL generalized model checking $M \models_{\preceq} \varphi$ is linear in M for weak and safety properties, and quadratic in M for response, persistence, and generalized reactivity[1] properties.*

Proof. From the proof of Theorem 6 it follows that we combine a deterministic automaton for the property with the model to get an APW over a 1-letter alphabet. From Theorem 14 it follows that if the LTL property is a safety or obligation property the DPW, and the resulting APW, are weak. If the LTL property is a response property, the DPW is in fact a DBW. If the LTL property is a persistence property, the DPW is in fact a DCW. If the LTL property is a generalized reactivity[1] property, the DPW has three

priorities. Recall that the APW is the product of the DPW and the model. Thus, the APW is linear in the size of the model. The desired upper bound now follows directly from Theorem 13. □

Note that LTL GMC for persistence properties can be solved in quadratic time in the size of the model, instead of in linear time as incorrectly stated in Theorem 5 of [GJ02]. The root cause of this error is the same as the one for Theorem 25 of [BG00], as the proofs of both theorems rely on the same product construction, now corrected in Theorem 6 of this paper.

Finally, we clarify a subtle misconception regarding generalized model checking of CTL properties. Given a CTL property, we can construct directly an NBT that is at most exponential in the size of the property that accepts all trees that satisfy the property [KVW00]. Generalized model checking can then be solved by combining this NBT with the model to obtain an ABW over a 1-letter alphabet [BG00]. According to Theorem 13 the emptiness of this ABW can be established in quadratic time. Thus, the complexity of GMC with respect to CTL properties is exponential in the formula and quadratic in the model, which is optimal [BG00]. As with LTL the quadratic complexity in the model follows from the type of acceptance condition used by the automaton for the formula. We are interested in classes of properties for which automata require simpler acceptance conditions. If the CTL property can be recognized by an NWT, the complexity in the size of the model reduces to linear. In the proof of Theorem 7 of [GJ02] it is assumed that if a CTL property can be recognized by an NCT then it can also be recognized by an NWT. However, it is currently unknown whether this is the case (cf. Section 6) and the proof of that theorem is therefore incomplete.

6 Conclusions

We study generalized model checking for linear time properties. We show that the classical definitions of GMC is 2EXPTIME-complete in the size of the formula and polynomial in the structure. We study a linear version of the completeness preorder and show that this preorder induces a GMC problem that is EXPSPACE-complete in the size of the formula. We then proceed to show that for interesting classes of properties the model complexity can be restricted to a low order polynomial.

We have presented our work in the framework of partial Kripke structures. Other equally expressive 3-valued models [GJ03] include Modal Transition Systems [LT88] and Kripke Modal Transition Systems [HJS01]. The complexity bounds given in this paper carry over to those closely related modeling formalisms.

The proof of Theorem 8 reduces realizability of LTL to GMC. The similarity actually goes in both directions. A GMC problem can be translated to a 2-person game where the specification (in LTL or in branching-time logic) can be translated to the winning condition. In a 2-person game players verifier and refuter alternate in moving a token along the edges of a graph. If the infinite path made by the token satisfies an LTL formula, verifier wins and otherwise she loses. If the winning condition is expressed in terms of branching-time logic, instead of considering a path in the graph, we consider the infinite unwinding of the game graph and prune the unwinding so that nodes that correspond to decisions of verifier have exactly one successor. The translation of the GMC problem

to such a game is as follows. The game graph itself is similar to the model, where decisions of the refuter correspond to the branching of the original model and decisions of the verifier correspond to the values given to undetermined propositions. The formula to be checked on the model is translated to the winning condition in the game. Much like the proofs of the lower bounds above, this straightforward translation may result in a game graph that is exponential in the number of propositions whose value is unknown. We can further reduce the number of nodes in the game graph to a product of the number of propositions whose value is unknown and the size of the model using the techniques in the proofs of Theorems 8 and 10. It may be possible to reduce the number of nodes in the game graph to a constant times the number of states of the model.

We have seen that for interesting classes of LTL and CTL properties the complexity in term of the model can be restricted to linear or quadratic. We classify the properties according to deterministic word automata and nondeterministic tree automata that match these formulas. While most popular types of properties are covered above, characterization of the exact classes of formulas that can be translated to these types of automata is an interesting problem. That is, what are the exact subsets of LTL that can be translated to DWW and to DBW? Is there a simple syntactic way to express these subsets? The same problem for CTL (and other branching-time logics) involves tree automata. For every CTL property there exist an NBT and an AWT recognizing the same set of trees [KVW00]. What CTL properties can be translated to NWT? Is there a syntactic way to express these subsets? We know that if a word language can be recognized by a DBW and by a DCW, then it can be recognized by a DWW [KMM04]. This suggests the following natural question: Given a tree language that is accepted by an NCT and by an NBT, can it be recognized by an NWT? From a practical point of view, it could be interesting to study the specific case of CTL properties that are recognized by NCT.

Acknowledgements. We thank Michael Huth for comments on an earlier version and Orna Kupferman for a discussion of the relative expressive power of NBT and NCT.

References

[AMPS98] Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: IFAC Symp. on System Structure and Control, pp. 469–474. Elsevier, Amsterdam (1998)

[BG99] Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)

[BG00] Bruns, G., Godefroid, P.: Generalized model checking: Reasoning about partial state spaces. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 168–182. Springer, Heidelberg (2000)

[BR01] Ball, T., Rajamani, S.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)

[GC05] Gurfinkel, A., Chechik, M.: How Thorough is Thorough Enough? In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 65–80. Springer, Heidelberg (2005)

[GH05] Godefroid, P., Huth, M.: Model Checking Vs. Generalized Model Checking: Semantic Minimizations for Temporal Logics. In: 20th Logic in Computer Science, pp. 158–167 (2005)

[GHJ01] Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based Model Checking using Modal Transition Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)

[GJ02] Godefroid, P., Jagadeesan, R.: Automatic Abstraction Using Generalized Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 137–150. Springer, Heidelberg (2002)

[GJ03] Godefroid, P., Jagadeesan, R.: On the Expressiveness of 3-Valued Models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 206–222. Springer, Heidelberg (2002)

[GS97] Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

[GTW02] Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)

[GWC06] Gurfinkel, A., Wei, O., Chechik, M.: Systematic Construction of Abstractions for Model-Checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 381–397. Springer, Heidelberg (2005)

[HJMS02] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: 29th Principles of Programming Languages, pp. 58–70 (2002)

[HJS01] 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)

[Jur00] Jurdziński, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)

[Kle87] Kleene, S.C.: Introduction to Metamathematics. North-Holland, Amsterdam (1987)

[KMM04] Kupferman, O., Morgenstern, G., Murano, A.: Typeness for ω-regular automata. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 324–338. Springer, Heidelberg (2004)

[KPP03] Kesten, Y., Piterman, N., Pnueli, A.: Bridging the gap between fair simulation and trace containment. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 381–393. Springer, Heidelberg (2003)

[KVW00] Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47(2), 312–360 (2000)

[LT88] Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: 3rd Logic in Computer Science, pp. 203–210 (1988)

[MP92] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1992)

[Pit07] Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Logical Methods in Computer Science 3(3), 5 (2007)

[PPS06] Piterman, N., Pnueli, A., Saar, 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)

[PR89] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: 16th Principles of Programming Languages, pp. 179–190 (1989)

[RW89] Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. IEEE Transactions on Control Theory 77, 81–98 (1989)

[Saf88] Safra, S.: On the complexity of ω-automata. In: 29th Foundations of Computer Science, pp. 319–327 (1988)

[SVW87] Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theoretical Computer Science 49, 217–237 (1987)

[vEB97] van Emde Boas, P.: The convenience of tilings. In: Complexity, Logic and Recursion Theory. Lecture Notes in Pure and Applied Mathetaics, vol. 187, pp. 331–363 (1997)

[VW86] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and System Science 32(2), 182–221 (1986)

[VW94] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

Monitoring the Full Range of ω-Regular Properties of Stochastic Systems*

Kalpana Gondi, Yogeshkumar Patel, and A. Prasad Sistla

University of Illinois at Chicago

Abstract. We present highly accurate deterministic, probabilistic and hybrid methods for monitoring the full range of ω-regular properties, specified as Streett automata, of stochastic systems modeled as Hidden Markov Chains. The deterministic algorithms employ timeouts that are set dynamically to achieve desired accuracy. The probabilistic algorithms employ coin tossing and can give highly accurate monitors when the system behavior is not known. The hybrid algorithms combine both these techniques. The monitoring algorithms have been implemented as a tool. The tool takes a high level description of an application with probabilities and also a Streett automaton that specifies the property to be monitored. It generates a monitor for monitoring computations of the application. Experimental results comparing the effectiveness of the different algorithms are presented.

1 Introduction

In the verification of concurrent and distributed programs, both safety and liveness properties play important roles. Liveness properties are proved correct assuming certain fairness/liveness properties of the environment that influence the execution of the concurrent programs. In the simplest of the cases, the environment is a scheduler that schedules the execution of the computation steps of the different processes, and one makes the weakest necessary fairness assumptions on the scheduler to prove the liveness properties. There are many other such assumptions one makes about fair delivery of messages in a distributed system, or about the failure of processes or about the release of acquired resources by users, etc. At run time, the verified liveness properties are going to be satisfied only if the assumed fairness properties are maintained. Thus, one has to either monitor for the violation of the assumed fairness properties, or monitor for the violation of liveness properties at run time. Monitoring the violations of the fairness properties may not be possible since the required information is not usually visible. As a consequence, monitoring the violation of liveness properties becomes necessary and important.

Monitoring of both safety and liveness properties also becomes necessary when one uses an existing off-the-shelf component that may not guarantee satisfaction

* This research is partly supported by the NSF grants CCF-0742686 and CCR-0205365.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 105–119, 2009.

of these properties in all executions; a prior verification of such components is not feasible due to unavailability of source code. Even in the cases where the component is found to be defective, we may want to use it hoping that the incorrect computations occur rarely. To handle the situation where such incorrect computation may occur, we need to monitor for such computations. Existing works [7,8,16] propose an approach for customizing such components to user requirements using a *conservative* run time monitor. In this scheme, one identifies a safety property that implies the given property f (in general, f is the intersection/conjunction of a safety and a liveness property).

In recent work [15], systems modeled as Hidden Markov Chains (HMC) were considered and conservative monitors were given for them. (HMCs are widely used formalisms to model systems whose state, at any time during an execution, can not be completely observed). A measure of accuracy was defined for conservative monitors called *acceptance* accuracy which is the probability that a good computation of the system (i.e., one satisfying the given property f) is accepted by the monitor. The method given in [15] only monitors for violations of properties specified by deterministic Buchi automata.

In this paper, we give conservative monitors that monitor for violations of properties specified by deterministic Streett automata which are more expressive than deterministic Buchi automata. (For example, for monitoring violations of liveness properties, such as strong fairness, we need Streett automata.) In a Streett automata each acceptance condition is given by a pair of sets of states $(RED, GREEN)$. Our method simulates the automaton on the output sequence generated by the system; it uses a counter/timeout with each acceptance pair. The timeout is reset whenever a $GREEN$ state appears; it is decremented when ever a RED state occurs. We show that by using a reset value for the counter that increase linearly with the number of $GREEN$ states, seen thus far, we can achieve accuracy arbitrarily close to 1. When the HMC is fully visible this monitor has accuracy 1. This result is quite surprising since earlier works [16] demonstrated that such a scheme, of using one counter with each accepting pair, is not complete for monitoring non-stochastic systems.

We also give a randomized monitor for monitoring HMCs for violations of properties given by Streett automaton. In these methods, whenever a RED state is seen then the computation is rejected with some low probability p. This probability of rejection remains constant until a $GREEN$ state is seen, and is reduced by a constant factor with each occurrence of a $GREEN$ state; thus, the probability of rejection decreases in geometric progression with the number of $GREEN$ states. We show that by choosing an appropriate initial probability and by choosing appropriate factor of reduction, we can achieve an accuracy that is arbitrarily close to 1. This result is quite interesting since earlier approaches had shown that, in general when the system behavior is non-stochastic and is not known in advance, then one cannot have a randomized monitor (called *strong monitors*) that accepts every good computation with non-zero probability.

Finally, we give a hybrid monitor that employs both counters as well as randomization. Such a monitor uses the counter not only as a time out for invoking

the random rejections, but also for generating extremely low values of probabilities. The method increments the reset value of the counter after each $GREEN$ state is seen. This monitor can also be tuned to achieve a desired accuracy as close to 1 as required.

In summary, the following are the main contributions of the paper:

- A conservative deterministic monitor employing counters, for monitoring violations of properties specified by deterministic Streett automata, for systems modeled as HMCs. These monitors can be designed to achieve a desired accuracy and have accuracy 1 when the HMC is fully visible.
- A probabilistic monitor which can also be tuned to achieve any desired accuracy, but requires generation of low probabilities.
- A hybrid monitor that uses counters as well as probabilistic rejections. It can also be designed to achieve any given accuracy. It generates low probabilities by using unbiased coin tosses.
- Experimental results showing the effectiveness of the methods.

We have developed a tool, called SM (Stochastic Monitor), that takes a high level description of a probabilistic synchronous concurrent program and a property specified as deterministic Streett automaton and that monitors if a given sequence of inputs violates the property specified by the automaton. The tool implements the three different monitoring algorithms one of which can be invoked through an appropriate option. The high level description of the concurrent program is only used to check for early acceptance or rejection of an input sequence.

The paper is organized as follows. Section 2 contains definitions. Section 3 presents a highly accurate deterministic monitor. Section 4 and 5 describe probabilistic and hybrid methods respectively. Section 6 presents experimental results. Section 7 has concluding remarks and comparison to related work. Proofs of theorems are left out and can be obtained from a full version of the paper on the last author's web site.

2 Definitions and Notation

Sequences. Let S be a finite set. Let $\sigma = s_0, s_1, \ldots$ be a possibly infinite sequence over S. The length of σ, denoted as $|\sigma|$, is defined to be the number of elements in σ if σ is finite, and ω otherwise. For any $i \geq 0$, $\sigma[0, i]$ denotes the prefix of σ up to s_i. If α_1 is a finite sequence and α_2 is either a finite or an ω-sequence then $\alpha_1\alpha_2$ denotes the concatenation of the two sequences in that order. We let S^*, S^ω denote the set of finite sequences and the set of infinite sequences over S. If $C \subseteq S^\omega$ and $\alpha \in S^*$ then αC denotes the set $\{\alpha\beta \ : \beta \in C\}$.

Automata. A *Streett automaton* (SA for short) $\mathcal{A}$ on infinite strings is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states; Σ is a finite alphabet of symbols; $\delta: Q \times \Sigma \to 2^Q$ is a transition function; $q_0 \in Q$ is an initial state; $F \subseteq 2^Q \times 2^Q$ is a set of pairs of subsets of states. Each pair $(RED, GREEN) \in F$ is called an *accepting pair* of $\mathcal{A}$. The *generalized transition function* $\delta^*: Q \times \Sigma^* \to 2^Q$

is defined in the usual way, i.e., for every state q, $\delta^*(q, \epsilon) = \{q\}$, and for any $\sigma \in \Sigma^*$ and $a \in \Sigma$, $\delta^*(q, \sigma a) = \cup_{q' \in \delta^*(q,\sigma)} \delta(q', a)$. If for every $(q, a) \in Q \times \Sigma$, $|\delta(q, a)| = 1$, then $\mathcal{A}$ is called a *deterministic* Streett automaton. Unless otherwise stated, we only consider deterministic automata. Let $\sigma : a_1, \ldots$ be an infinite sequence over Σ. A *run* r *of* $\mathcal{A}$ on σ is an infinite sequence $r_0, r_1, \ldots$ over Q such that $r_0 = q_0$ and for every $i > 0$, $r_i \in \delta(r_{i-1}, a_i)$. The run r of a Streett automaton is *accepting* if for every accepting pair $(RED, GREEN)$ in F the following condition holds: if there exists an infinite set I of indices such that, for each $i \in I$, $r_i \in RED$ then there also exists an infinite set J of indices such that for each $i \in J$, $r_i \in GREEN$. The automaton $\mathcal{A}$ *accepts* the ω-string σ if its unique run over σ is an accepting run. The *language accepted by* $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of ω-strings that $\mathcal{A}$ accepts. A language L' is called ω*-regular* if it is accepted by some Streett automaton.

Hidden Markov Chains. We assume that the reader is familiar with basic probability theory and random variables and Markov chains. We consider stochastic systems given as Markov Chains [9] and monitor their computations for satisfaction of a given property specified by an automaton or a temporal formula. A Markov chain $G = (S, R, \phi)$ is a triple satisfying the following: S is a set of countable states; $R \subseteq S \times S$ is a total binary relation (i.e., for every $s \in S$, there exists some $t \in S$ such that $(s, t) \in R$); and $\phi : R \to (0, 1]$ is a probability function such that for each $s \in S$, $\sum_{(s,t)\in R} \phi((s, t)) = 1$. Note that, for every $(s, t) \in R$, $\phi((s, t))$ is non-zero. Intuitively, if at any time the system is in a state $s \in S$, then in one step, it goes to some state t such that $(s, t) \in R$ with probability $\phi((s, t))$. A finite path p of G is a sequence $s_0, s_1, ..., s_n$ of states such that $(s_i, s_{i+1}) \in R$ for $0 \leq i < n$. We extend the probability function to such paths, by defining $\phi(p) = \prod_{0 \leq i < n} \phi((s_i, s_{i+1}))$.

We assume that there is a finite set $\mathcal{P}$ of atomic propositions that represent conditions on system states. Let Σ denote $2^{\mathcal{P}}$, the power set of $\mathcal{P}$. Each member of Σ denotes the set of atomic propositions that are true in a state of the system. From here onwards, we assume that Σ is the input alphabet of the property automata that we consider. If the property is given by a temporal formula then the atomic propositions appearing in the formula are drawn from $\mathcal{P}$. For any $C \subseteq \Sigma^\omega$, let $\bar{C}$ denote the set $\Sigma^\omega - C$. For an atomic proposition $P \in \mathcal{P}$, when used in a sequence, P represents the set of elements of Σ that contain P; similarly $\neg P$ represents the set of elements that do not contain P.

A Hidden Markov Chain (HMC) [2] $H = (G, O, r_0)$ is a triple where $G = (S, R, \phi)$ is a Markov chain, $O : S \to \Sigma$ is the output function and $r_0 \in S$ is the initial state. Intuitively, for any $s \in S$, $O(s)$ is the output generated in state s and is the set of atomic propositions true in s; this output is generated when ever a transition entering state s is taken. The generated symbols become inputs to the monitor. H is called Hidden Markov chain because, one only observes the outputs generated in each state but not the actual state. We extend the output function O to paths of G as follows. For any finite path $p = s_0, s_1, ..., s_n$ in G, $O(p) = O(s_0), O(s_1), ..., O(s_n)$. The probability distribution on the single step state transition of G induces a probability distribution on the sets of sequences of

outputs generated. To define these distributions formally, let $\mathcal{E}$ be the smallest class of subsets of Σ^ω satisfying the following properties: for every $\alpha \in \Sigma^*$, $\alpha\Sigma^\omega \in \mathcal{E}$; $\mathcal{E}$ is closed under countable union (i.e., if $C_0, ...C_i, ...$ is a finite or infinite sequence of elements in $\mathcal{E}$, then $\bigcup_{i\geq 0} C_i$ is also in $\mathcal{E}$); it is closed under complementation (i.e., for every $C \in \mathcal{E}$, $\bar{C}$ is also in $\mathcal{E}$). The elements of $\mathcal{E}$ are called *measurable* subsets of Σ^ω. It is not difficult to see that $\mathcal{E}$ is also closed under countable intersections. It can be shown that, for any automaton $\mathcal{A}$ with input alphabet Σ, $L(\mathcal{A})$ is measurable [17].

Now, for any system state $r \in S$, we define a probability function $\mathcal{F}_r$ defined on $\mathcal{E}$ as follows. Intuitively, for any $C \in \mathcal{E}$, $\mathcal{F}_r(C)$ denotes the probability that an output sequence generated from the system state r, is in C. $\mathcal{F}_r$ is the unique probability measure satisfying all the probability axioms [9], such that for every $\alpha \in \Sigma^*$ and $C = \alpha\Sigma^\omega$, $\mathcal{F}_r(C)$ is the sum of $\phi(p)$, for all finite paths p of G starting from the state r such that $O(p) = \alpha$. For the HMC chain given in figure 1 and for $\alpha = (\{Q\}, \{Q\}, \{Q\})$, there are three paths p, i.e., s_0, s_0, s_0 and s_0, s_0, s_2 and s_0, s_2, s_2 such that $O(p) = \alpha$ and hence $\mathcal{F}_{s_0}(C) = \frac{5}{9}$.

Let $D \in \mathcal{E}$ be such that $\mathcal{F}_r(D) \neq 0$. We let $\mathcal{F}_{r|D}$ denote the conditional probability function given D; formally, for any $C \in \mathcal{E}$, $\mathcal{F}_{r|D}(C) = \frac{\mathcal{F}_r(C \cap D)}{\mathcal{F}_r(D)}$. For any LTL formula g, we let $\mathcal{F}_{r|g}$ denote the conditional distribution $\mathcal{F}_{r|D}$ where D is the set of input sequences that satisfy g. For any $\alpha \in \Sigma^*$ and $C = \alpha\Sigma^\omega$, we let $\mathcal{F}_r(\alpha)$ denote the probability $\mathcal{F}_r(C)$ and $\mathcal{F}_{r|\alpha}$ denote the conditional probability function $\mathcal{F}_{r|C}$. For a set $C \subseteq \Sigma^*$, we let $\mathcal{F}_r(C)$ denote $\mathcal{F}_r(C\Sigma^\omega)$.

Example 1. Consider the HMC S_1 given in figure 1. Here the set of atomic propositions $\mathcal{P} = \{P, Q\}$. It should be easy to see that $\mathcal{F}_{s_0}(\Diamond P) = \frac{1}{2}$.

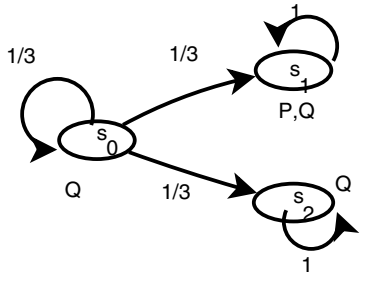


Fig. 1. System S_1

Deterministic Monitors. A monitor $M : \Sigma^* \to \{0, 1\}$ is a computable function with the property that, for any $\alpha \in \Sigma^*$, if $M(\alpha) = 0$ then $M(\alpha\beta) = 0$ for every $\beta \in \Sigma^*$. For an $\alpha \in \Sigma^*$, we say that M rejects α, if $M(\alpha) = 0$, otherwise we say M accepts α. Thus if M rejects α then it rejects all its extensions. For an infinite sequence $\sigma \in \Sigma^\omega$, we say that M rejects σ iff there exists a prefix α of σ that is rejected by M; we say M accepts σ if it does not reject it. Let $L(M)$ denote the set of infinite sequences accepted by M. It is not difficult to see that $L(M)$ is a safety property and is measurable. The *acceptance accuracy* of M for $\mathcal{A}$ with respect to the HMC H is defined to be the probability $\mathcal{F}_{r_0|L(\mathcal{A})}(L(M))$ where

r_0 is the initial state of H. Intuitively, it is the conditional probability that a sequence generated by the system is accepted by M, given that it is in $L(\mathcal{A})$. Roughly speaking, it is the fraction of the sequences in $L(\mathcal{A})$, generated from r_0, that are accepted by M. Let C, D be the complements of $L(\mathcal{A})$ and $L(M)$ respectively, i.e., $C = \Sigma^\omega - L(\mathcal{A})$ and $D = \Sigma^\omega - L(M)$. Then the *rejection accuracy* of M for $\mathcal{A}$ (also for $L(\mathcal{A})$) with respect to H is defined to be the probability $\mathcal{F}_{r_0|C}(D)$. This is the probability that a sequence generated by the system is rejected by M, given that it is not in $L(\mathcal{A})$.

We say that M is a *conservative monitor* for a language $L' \subseteq \Sigma^\omega$, if $L(M) \subseteq L'$, i.e., it rejects every sequence not in L'. We say that M is a conservative monitor for an automaton $\mathcal{A}$ (resp., for a LTL formula ϕ) if it is a conservative monitor for $L(\mathcal{A})$ (resp., for C where C is the set of sequences that satisfy ϕ). Note that the rejection accuracy of a conservative monitor is 1.

Example 2. Consider the following conservative monitor M_1 for the LTL formula $\Diamond P$. It accepts all finite sequences of length ≤ 2. It accepts a finite sequence of length greater than two only if it has a P in the first three symbols. Clearly, $L(M_1) = \{(\neg P)^i P\beta : i \leq 2, \beta \in \Sigma^\omega\}$. Now consider the system HMC of Example 1. The first input produced by S_1 from state s_0 is $\neg P$, the probability that either the second or the third symbol is a P is $\frac{4}{9}$. From this, it should be easy to see that $F_{s_0|\Diamond P}(L(M_1)) = \frac{8}{9}$. Hence the acceptance accuracy of M_1 for $\Diamond P$ with respect to the system S_1 is $\frac{8}{9}$.

Probabilistic Monitors. We also define probabilistic monitors. A *probabilistic monitor* $M : \Sigma^* \rightarrow [0, 1]$ is a function that associates a probability $M(\alpha)$ with each $\alpha \in \Sigma^*$ such that for every $\alpha, \beta \in \Sigma^*$, $M(\alpha\beta) \leq M(\alpha)$. Intuitively, $M(\alpha)$ denotes the probability that α is accepted by M. We extend M to infinite sequences as follows. For any $\sigma \in \Sigma^\omega$, $M(\sigma) = \lim_{i\rightarrow\infty} M(\sigma[0, i])$. $M(\sigma)$ represents the probability of acceptance of σ by M. We say that M is a probabilistic monitor for a language L' if $M(\sigma) = 0$ for all $\sigma \in \Sigma^\omega - L'$. That is every sequence not in L' is rejected with probability 1. Although we defined monitors as functions, many times monitors are given by algorithms (deterministic or probabilistic) that take inputs and reject some input sequences. With each such algorithm there is an implicitly defined unique monitor function.

The definition of acceptance accuracy for probabilistic monitors is a little more involved. Let M be a probabilistic monitor for an automaton $\mathcal{A}$ with input alphabet Σ. Let $\mathcal{F}_{r_0|L(\mathcal{A})}$ be the conditional probability distribution function on the set of input sequences generated by the system. For any $n > 0$, let $Y_n = \sum_{\alpha\in\Sigma^n}(\mathcal{F}_{r_0|L(\mathcal{A})}(\alpha)M(\alpha))$; note Σ^n is the set of all sequences of length n. Because of the monotonicity of the function M, it is easy to see that $Y_n \geq Y_{n+1}$ for all $n > 0$. We define the acceptance accuracy of M for $\mathcal{A}$ with respect to the given system to be $\lim_{n\rightarrow\infty} Y_n$. The rejection accuracy of M for $\mathcal{A}$ with respect to the HMC H, is defined to be the $\lim_{n\rightarrow\infty} Z_n$ where Z_n is obtained from the expression for Y_n by replacing $L(\mathcal{A})$ by its complement and $M(\alpha)$ by $1 - M(\alpha)$.

Example 3. Consider the following probabilistic monitor M_2 for $\Diamond P$. M_2 looks at the current symbol. If it is a P it accepts and it will accept all subsequent

inputs. If the current symbol is $\neg P$ and it has not seen a P since the beginning, then it rejects with probability $\frac{1}{3}$ and accepts with probability $\frac{2}{3}$. Formally, $M_2((\neg P)^{n-1}P\beta) = (\frac{2}{3})^{n-1}$ for all $n \geq 1$ and for all $\beta \in \Sigma^*$. Now consider the HMC S_1 given in Example 1. It can be shown by simple probabilistic analysis that $Y_n = 2\sum_1^{n-1}(\frac{2}{3})^i(\frac{1}{3})^i$. From this, we see that the accuracy of M_2 for $\Diamond P$ with respect to system S_1 , which is $\lim_{n\to\infty} Y_n$, is $\frac{4}{7}$. In general if the rejection probability at each step used is p, it can be shown that the acceptance accuracy of M_2 for $\Diamond P$ with respect to S_1 is $2\frac{1-p}{2+p}$. Thus the acceptance accuracy can be increased arbitrarily close to 1 by decreasing p.

3 Accurate Deterministic Monitors

In this section we give methods for designing highly accurate deterministic monitors that monitor the executions of a system, given as a HMC, against a property specified by a deterministic finite state Streett automaton. Here we assume that we know the initial system state and the output sequence generated by the system, but we can not observe the system state. All the monitors presented in the rest of the paper have a rejection accuracy of one. In the rest of the paper, we use the term accuracy of a monitor to simply refer to its acceptance accuracy.

Let $H = (G, O, r_0)$ be a HMC where $G = (S, R, \phi)$ is a finite Markov chain and $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be the given deterministic finite state Streett automaton. Let $(RED_j, GREEN_j)$, for $j = 1, .., l$ be the acceptance pairs in F. For any $q \in Q$, let $\mathcal{A}_q$ be the automaton obtained by changing the initial state of $\mathcal{A}$ to q. Let $(s, q) \in S \times Q$. We call each such pair a product state. Note that, for any such (s, q), $\mathcal{F}_s(L(\mathcal{A}_q))$ is the probability that an infinite output sequence generated from the system state s is accepted by the automaton $\mathcal{A}$ when started in the state q. We say that (s, q) is an *accepting* product state if $\mathcal{F}_s(L(\mathcal{A}_q)) = 1$. We say that it is a *rejecting* product state if $\mathcal{F}_s(L(\mathcal{A}_q)) = 0$. Later we show how the accepting and rejecting product states can be determined.

Our monitoring algorithm works as follows. As the monitor gets inputs from the system, it simulates the automaton $\mathcal{A}$ on the input sequence using the variable *a_state*. It also keeps a set *s_states* which is the set of states the system can be in. If at any time all states in *s_states* $\times$ {*a_state*} are accepting product states then it accepts. If all these states are rejecting product states then it rejects. Otherwise, it continues. In addition, for each acceptance pair, i.e., for each $j = 1, ..., k$, it maintains a counter, denoted by $counter_j$ variable, which is initialized to some value; it also maintains a variable i_j that gives the number of times a $GREEN_j$ state is encountered. In each iteration, for each j, $counter_j$ is decremented if the automaton state belongs to $RED_j - GREEN_j$. If the automaton state is a state in $GREEN_j$ then the $counter_j$ is reset. For any j, if $counter_j$ is zero before a $GREEN_j$ state is reached then, it rejects. The variable $counter_j$ is initialized and reset to a value given by the function $f_j()$ as shown in the algorithm. The formal description of the algorithm for the monitor is given in figure 2. The procedure GetInput-and-Update() updates the automaton state, gets next input and updates the variable *s_states*. It also checks if the input can

$a\_state := q_0$; $s\_states := \{r_0\}$;
for $j := 1$ *to* l $\{i_j := 1$; $counter_j := f_j(s\_states, i, a\_state)\}$;
$x := O(r_0)$;
Loop forever
{ GetInput-and-Update();
  for $j := 1$ to l
    If $a\_state \in RED_j - GREEN_j$ **then** $counter_j := counter_j - 1$;
    If $counter_j = 0$ and $a\_state \notin GREEN_j$ **then** reject();
    If $a\_state \in GREEN_j$ **then**
      $\{i_j := i_j + 1$; $counter_j := f_j(s\_states, i_j, a\_state)\}$
}
Procedure GetInput-and-Update()
{ $a\_state := \delta(a\_state, x)$;
  $x := get\_nextinput()$;
  $s\_states := \{s' : (s, s') \in R,\ s \in s\_states,\ O(s') = x\}$;
  If every state in $s\_states \times \{a\_state\}$ is an accepting product state
    then accept();
  If every state in $s\_states \times \{a\_state\}$ is a rejecting product state
    then reject();
}

Fig. 2. Deterministic Algorithm

be accepted or rejected immediately as explained earlier. The variable x denotes the current input symbol and the variable i_j records the number of times the $counter_j$ has been reset. Here r_0 is the initial state of the system. Whenever the monitor rejects (or accepts) then it immediately stops; in this case, it is assumed that it rejects (or accepts) all future inputs.

It is easy to see that the monitor rejects any input sequence that is not in $L(\mathcal{A})$ since after a certain point, for some j, a_state is in RED_j infinitely often, but is not in $GREEN_j$ from that point and $counter_j$ counts down to zero.

The accuracy of the monitor is highly dependent on the functions f_j used in resetting the counter. One possibility is to reset it to a constant k. In this case, it can be shown that the accuracy of the resulting monitor is going to be zero many times (figure 3 is one such example). The following theorem shows that by increasing the reset value of $counter_j$ linearly with i_j, we can achieve a desired accuracy.

Theorem 1. *For any given rational y such that $0 \leq y \leq 1$, for each $j = 1, ..., k$, there exists a constant a_j such that if $f_j(X, i, q) = a_j \cdot i$, for every $X \subseteq S$ and $q \in Q$, then the accuracy of the above monitor given in figure 2 is at least y. Further more such constant a_j is computable in time polynomial in the sizes of $H, \mathcal{A}$.*

For an interesting subclass of HMCs, called *fully visible* HMCs, we can obtain a much simpler monitoring algorithm that only executes the procedure GetInput-

and-Update in the loop body. This simpler algorithm has accuracy 1 [15]. (HMC $H = (G, O, r_0)$ is said to be *fully visible* if O is a one-one function).

Example 4. Now consider the example of a HMC given in figure 3. In this example, initially the system is in state s which is a non-critical region (labeled by the proposition N). From s, it may loop there or go to s' or go to state t which is the trying region where the user sends a request for the resource, labeled by T. From here, the system may loop, or go to state v denoting that the resource server crashed or go to state w where the resource is granted. Note both states t, v are labeled by T. Thus we can not tell whether the system is in state t or in v. In state s', it can loop or go to t' where it requests for the resource. In t', it may loop or go to w'. In state w' the resource is allocated. Note that the resource server does not crash when requested from t'. This is because, here, a more reliable server is employed. Now our monitoring algorithm can be used to monitor for the desired LTL property $g = \Box\Diamond T \rightarrow \Box\Diamond C$. We express this property using a Streett automaton with an acceptance condition consisting of one pair.

Let $\mathcal{G}(x)$ be the function given by $\mathcal{G}(x) = \prod_{i\geq 1}^{\infty}(1-\frac{1}{x^i})$. It is easy to show that $\mathcal{G}(x) > 0$ for all integer $x \geq 2$ and monotonically increases with x approaching 1 as x goes to ∞. Let y, $0 \leq y \leq 1$, be the desired accuracy and a be the smallest integer such that $a \geq 2$ and $\mathcal{G}(a) \geq y$. It can be shown that, when we use $a \cdot i$ to be the function $f_1(X, i, q)$, the monitor given in figure 2 has an accuracy greater than or equal to y for the HMC given in figure 3.

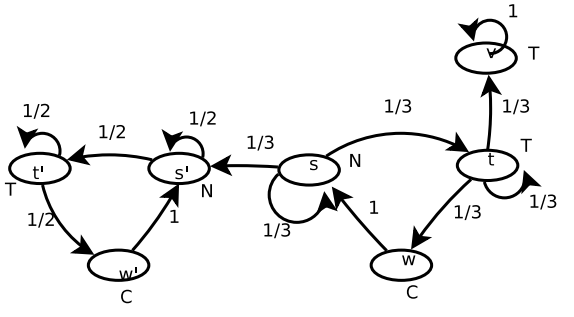


Fig. 3. Resource Acquisition

4 Probabilistic Monitors

In this section, we present a probabilistic monitor that monitors a system modeled as a HMC for violations of properties specified by deterministic Streett automata. As given in section 3, let $H = (G, O, r_0)$ be a HMC where $G = (S, R, \phi)$ is a finite Markov chain and $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be the given deterministic finite state Streett automaton. Let $(RED_j, GREEN_j)$, for $j = 1, .., l$ be the acceptance pairs in F. For any $q \in Q$, let $\mathcal{A}_q$ be the automaton obtained by changing the initial state of $\mathcal{A}$ to q.

As before, our algorithm simulates the automaton $\mathcal{A}$ on the output sequence generated by the system. For each j, $1 \leq j \leq l$, it rejects with some probability whenever it sees a state in $RED_j - GREEN_j$; this probability of rejection remains constant until a $GREEN_j$ node is seen at which point this probability of rejection is decreased by a constant factor c. Thus, it is easy to see that if we have infinite number of RED_j automaton states but only a finite number of $GREEN_j$ states (for some j) then the algorithm rejects with the same probability infinitely often and hence computation is eventually rejected with probability 1. This algorithm may reject some good computations with some non-zero probability. However, as we show, the accuracy of the algorithm can be made as close to 1 as possible by choosing an appropriate value of $c \geq 2$ and by choosing appropriate initial probabilities.

The detailed description of the algorithm is given below. The function GetInput-and-Update() is as given in section 3. The algorithm maintains variables $p_1, ..., p_l$ which are the probabilities of rejection one for each of the acceptance pairs. These values are decreased by a constant factor c, each time the corresponding $GREEN$ state is seen.

$a\_state := q_0$; $s\_states := \{r_0\}$;
for $j := 1$ *to* l *initialize* p_j *to* μ_j;
$x := O(r_0)$;
Loop forever
{ GetInput-and-Update();
  for $j := 1$ to l
    If $a\_state \in RED_j - GREEN_j$ **then** reject with probability p_j;
    If $a\_state \in GREEN_j$ **then** $p_j := \frac{p_j}{c}$
}

Fig. 4. Probabilistic Algorithm

In the probabilistic algorithm shown in Figure 4, it is possible that the computation may be rejected after an input with different probabilities for different values of j.

Theorem 2. *For each $y \in (0, 1)$, there exists a constant $c \geq 2$, and an initial probability value μ_j for each j, $1 \leq j \leq l$, so that the accuracy of the algorithm in figure 4 is at least y. Further more, the above constants can be computed efficiently from G and $\mathcal{A}$.*

5 Hybrid Monitors

The hybrid algorithm is a combination of the deterministic and probabilistic algorithms. As before, for each $j = 1, ..., l$, we maintain a variable $counter_j$ corresponding to the acceptance pair $(RED_j, GREEN_j)$. The algorithm also maintains a variable $prev_j$ which denotes the current reset counter value.

```
a_state := q0; s_states := {r0};
 for j := 1 to l {flag_j := true; initialize counter_j; prev_j := counter_j};
x := O(r0);
Loop forever
{ GetInput-and-Update();
    for j := 1 to l
        If a_state ∈ RED_j − GREEN_j then
            counter_j := counter_j − 1;
            Toss a fair coin and if "tails" then set flag_j to false;
        If counter_j = 0 and a_state ∉ GREEN_j then
            If flag_j then reject()
            else
              counter_j := prev_j; flag_j := true;
        If a_state ∈ GREEN_j then
            {flag_j := true; counter_j := prev_j := prev_j + 1}
}
```

Fig. 5. Hybrid Algorithm

In addition, for each such j, we also maintain a boolean variable $flag_j$. This variable is initially set to *true* and is also set to *true* whenever a state from $GREEN_j$ is seen. Whenever a state in $RED_j - GREEN_j$ is seen, $counter_j$ is decremented and an unbiased coin is tossed to set the $flag_j$ to $false$ if the coin turns "tails". When a $GREEN_j$ state is seen, the corresponding counter is reset to the next higher value (which is greater than the previous value by 1). Whenever the value of $counter_j$ is zero and the current state is not a $GREEN_j$ state and $flag_j$ is *true* then the algorithm rejects. Note that when $counter_j$ has value zero then the value of $flag_j$ is true only if the last k_j coin tosses, corresponding to the j^{th} acceptance pair, have all turned "heads"; the probability of this happening is $\frac{1}{2^{k_j}}$; here k_j is the latest reset value to which $counter_j$ was reset. The formal description of the algorithm is given in figure 5. The function GetInput-and-Update is as given in section 3.

Theorem 3. *For every $y \in (0, 1)$, there exists an initial counter value such that the accuracy of the algorithm given in figure 5 is at least y.*

6 Experimental Results

We have developed a tool, called SM (Stochastic Monitor), that implemented all the three monitoring algorithms one of which can be invoked by an input option with appropriate parameters. The proposed monitoring algorithms required us to compute accepting and rejecting product states. These two sets of states are computed by SM as follows. SM takes the high level description of the synchronous probabilistic concurrent program $\mathcal{P}$ whose outputs SM is supposed to monitor. From $\mathcal{P}$, SM constructs a HMC $\mathcal{M}$ modeling its operation. This construction is achieved using the PRISM [13] tool. After this, it constructs the

product of $\mathcal{M}$ and $\mathcal{A}$, giving a product Markov chain $\mathcal{M}'$. Recall, from section 3, that the product state (s, q) is an accepting product state if $\mathcal{F}_s(L(\mathcal{A}_q)) = 1$ and is a *rejecting* product state if $\mathcal{F}_s(L(\mathcal{A}_q)) = 0$. We can consider the product Markov chain as a directed graph. A Strongly Connected Component (SCC) C of $\mathcal{M}'$ is called an accepting SCC if for each accepting pair $(RED, GREEN)$ of $\mathcal{A}$ the following condition holds: if C has a product state of the form (s, q) for some $q \in RED$ then it also has a product state of the form (s', q') for some $q' \in GREEN$. A SCC is called a *terminal* SCC if no other SCC is reachable from it. It can be shown that a product state is an accepting product state (resp., rejecting product state) iff all terminal strongly connected components reachable from it are accepting (resp., non-accepting) strongly connected components. Using these properties, the accepting and rejecting product states are computed using the standard graph algorithms that take only time linear in the size of $\mathcal{M}'$. The computed accepting, rejecting product states are used in the procedure GetInput-and-Update() given in the algorithms.

We have tested all three algorithms with different parameters on a system that implemented a modified version of the classic Peterson's mutual exclusion algorithm. This system consists of two processes $0, 1$ that repeatedly try to get into the mutual exclusion region. The system we considered operates in two modes. In the first mode, we allowed process 1 to die in the critical section. We modeled the operation of the algorithm as a HMC. The output of each state indicates whether process 0 is in non-critical section, in trying section or in critical section. From any global state one of the processes is chosen for executing the next step with probability $\frac{1}{2}$; however when process 1 is in critical section, we add an additional failure transition to a failure state with some small probability, say 0.1. In addition, when both the processes are in non-critical section the system can change from the first mode to the second mode with probability 0.4. Once in the second mode, the system remains in that mode. In the second mode, both processes operate as before except that neither of the processes fail in this mode. For this system we monitored the following property specified as a Streett automaton: if process 0 is in trying section infinitely often then it should be in critical section infinitely often. Note that this property holds with probability less than 1.

We formulated the above system as a synchronous system with two user processes and a scheduler. At each step the scheduler probabilistically schedules one of the two processes with equal probability and the scheduled process executes one step. The scheduler transitions are fused with the transitions of the user processes so that effectively we only have two processes. The resulting processes are specified using the input language of PRISM.

We tested all the three monitoring algorithms on this example and computed their accuracies. We generated the input sequences to the monitoring algorithms by simulating the HMC. Each input sequence is of length 10^5. If the sequence is not rejected during the whole input sequence then it is considered as accepted. A sequence can also be accepted explicitly as given in the monitoring algorithm. Any rejection due to the expiry of a counter (in the deterministic algorithm) or due to a probabilistic choice (in the probabilistic/hybrid algorithm)

Table 1. Peterson's Mutal Exclusion Algorithm

Deterministic			**Probabilistic**			**Hybrid**		
Counter	Accuracy (%)	Time (ms)	Prob.	Accuracy (%)	Time (ms)	Counter	Accuracy (%)	Time (ms)
4	16	130.27	0.25	23	151.30	4	80	1580.21
8	33	386.43	0.13	30	408.69	8	96	2816.90
16	50	770.02	0.06	50	827.06	16	100	3179.77
32	53	771.65	0.03	63	1796.08	24	100	2909.28

Table 2. The Bounded Retransmission Protocol with N=2 and Max=4

Deterministic			**Probabilistic**			**Hybrid**		
Counter	Accuracy (%)	Time (ms)	Prob.	Accuracy (%)	Time (ms)	Counter	Accuracy (%)	Time (ms)
12	10	0.31	0.13	16	0.14	3	50	0.39
16	63	0.45	0.06	40	0.34	4	80	0.40
20	93	0.47	0.03	66	0.44	6	96	0.51
24	96	0.51	0.02	76	0.44	8	100	0.62

is considered as a false rejection. The accuracies are computed using these statistics. We computed the accuracy by taking 30 such sequences and taking their average accuracy. These accuracies are reported in table 1. For this example, the monitored property can also be specified by a deterministic Buchi automaton. We also tested our tool with another mutual exclusion example in which two processes repeatedly try to enter a critical section using semaphores. For this example, we needed deterministic Streett automaton for specifying the property to be monitored. Experimental results for this example are similar and are left out. We also evaluated the Bounded Retransmission Protocol[3] using our SM tool. This is a network transmission protocol, where a file or message can be sent in a few chunks and each chunk can be retransmitted only in a fixed number of times. The retransmission would be required in case of any packet loss in the channel. We monitored the behavior of the sender. The property that, eventually the sender successfully sends the file (or message) is monitored. We modified the specification of the PRISM input so that we can monitor only the sender. The results are shown in the following table 2. In bot the examples, for all the three monitoring algorithms, we see that as the accuracies increase the time taken by the monitor also increases. We also see that the Hybrid algorithm gives high accuracy even with small counter values. It may appear that the monitor time, i.e., the overhead, for the hybrid algorithm is higher. On a close examination, we see that for the same accuracy, the overhead for the hybrid algorithm is similar to that of the other algorithms. For example in bounded retransmission protocol, the deterministic algorithm gives an accuracy of 96 for a counter of 24 and it takes 0.51 ms to monitor. Similarly, the Hybrid algorithm gives 96% accuracy for a counter of 6, but the time it takes is same as that of deterministic.

Our experiments indicate that hybrid algorithms achieve high levels of accuracy using small counter values with slightly higher overhead due to randomization.

7 Conclusion and Related Work

As has been indicated in the introduction, earlier works [7,8,16] gave different techniques for synthesizing conservative deterministic monitors for monitoring temporal properties. Deterministic monitors for monitoring Buchi properties of HMCs are given in [15]. In this paper, we have given deterministic, probabilistic and hybrid monitors for monitoring properties of HMCs specified by Streett automata. We have also shown that any given accuracy less than 1 can be achieved, and have given efficient techniques for initializing the parameters of the monitor to achieve that accuracy. As part of future work, we need to check the effectiveness of these algorithms on real world problems.

Deterministic liberal monitoring algorithms have been proposed in [1]. Run time monitoring has also been used for interface synthesis in [10] where the interactions between the module and the interface are considered as a two person game.

In [5,6] Larsen et al. propose a method which, given a context specification and an overall specification, derive a temporal safety property characterizing the set of all implementations which, together with the given context, satisfy the overall specification. There has been much work in the literature on monitoring violations of safety properties in distributed systems. In these works, the safety property is typically explicitly specified by the user. A method for monitoring and checking quantitative and probabilistic properties of real-time systems has been given in [14]. These works take specifications in a probabilistic temporal logic (called CSL) and monitors for its satisfaction. The probabilities are deduced from given the repeated occurrence of events in a computation.

None of the above works employ accuracy measures for monitors and none of them use randomization for monitoring liveness properties as we do. Our techniques are entirely new and have been experimentally validated.

Model checking probabilistic systems modeled as Markov chains was considered in the works of [4,11,12,17]. Some of these works construct a product of the Markov chain and the automata/tableaux associated with the LTL formula for performing the verification. PRISM [13] and ETMCC [4] are two popular tools that can be used for model checking finite state Markov chains, Continuous time Markov chains and Markov Decision Processes. While all of them concentrate on verification, we concentrate on the corresponding monitoring problem. Further more, we assume that during the computation the state of the system is not fully visible.

References

1. Amorium, M., Rosu, G.: Efficient monitoring of omega-languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)

2. Cappe, O., Moulines, E., Riden, T.: Inferencing in Hidden Markov Models. Springer, Heidelberg (2005)
3. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 127–165. Springer, Heidelberg (1994)
4. Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., Siegle, M.: A markov chain model checker. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 347–362. Springer, Heidelberg (2000)
5. Larsen, K.: Ideal Specification Formalisms = Expressivity + Compositionality + Decidability + Testability +... In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458. Springer, Heidelberg (1990)
6. Larsen, K.: The expressive power of implicit specifications. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510. Springer, Heidelberg (1991)
7. Margaria, T., Sistla, A.P., Steffen, B., Zuck, L.D.: Taming interface specifications. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 548–561. Springer, Heidelberg (2005)
8. Margaria, T., Sistla, A., Steffen, B., Zuck, L.D.: Taming interface specifications (2005), `www.cs.uic.edu/~sistla`
9. Papoulis, A., Pillai, S.U.: Probability, Random Variables and Stochastic Processes. McGrawHill, NewYork (2002)
10. Pnueli, A., Zaks, A., Zuck, L.D.: Monitoring interfaces for faults. In: Proceedings of the 5th Workshop on Runtime Verification (RV 2005) (2005) (to appear in a special issue of ENTCS)
11. Pnueli, A., Zuck, L.: Probabilistic verification by tableux. In: Proceedings of First IEEE Symposium on Logic in Computer Science, pp. 322–331 (1986)
12. Pnueli, A., Zuck, L.: Probabilistic verification. Information and Computation 103, 1–29 (1993)
13. Probabilistic Model Checker, `http://www.prismmodelchecker.org`
14. Sammapun, U., Lee, I., Sokolsky, O.: Rt-mac:runtime monitoring and checking of quantitative and probabilistic properties. In: Proc. of 11th IEEE International Conference on Embedded and Real-time Computing Systems and Applications (RTCSA 2005), pp. 147–153 (2005)
15. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008)
16. Sistla, A.P., Zhou, M., Zuck, L.D.: Monitoring off-the-shelf components. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 222–236. Springer, Heidelberg (2005)
17. Vardi, M.: Automatic verification of probabilistic concurrent systems. In: 26th annual Symposium on Foundations of Computer Science, pp. 327–338. IEEE Computer Society Press, Los Alamitos (1985)

Constraint-Based Invariant Inference over Predicate Abstraction

Sumit Gulwani[1], Saurabh Srivastava[2,⋆], and Ramarathnam Venkatesan[3]

[1] Microsoft Research, Redmond
sumitg@microsoft.com
[2] University of Maryland, College Park
saurabhs@cs.umd.edu
[3] Microsoft Research, Redmond
venkie@microsoft.com

Abstract. This paper describes a constraint-based invariant generation technique for proving the validity of safety assertions over the domain of predicate abstraction in an interprocedural setting. The key idea of the technique is to represent each invariant in bounded DNF form by means of boolean indicator variables, one for each predicate p and each disjunct d denoting whether p is present in d or not. The verification condition of the program is then encoded by means of a boolean formula over these boolean indicator variables such that any satisfying assignment to the formula yields the inductive invariants for proving the validity of given program assertions.

This paper also describes how to use the constraint-based methodology for generating maximally-weak preconditions for safety assertions. An interesting application of maximally-weak precondition generation is to produce maximally-general counterexamples for safety assertions. We also present preliminary experimental evidence demonstrating the feasibility of this technique.

1 Introduction

Predicate abstraction [1] is a commonly used technique for proving program properties. This involves over-approximating the set of reachable states of the program using formulas with boolean structure over a given set of predicates. This over-approximation is usually computed using fixed-point based techniques like abstract interpretation or model checking. One of the main advantages of the predicate abstraction domain is that it can represent disjunctions as opposed to other abstract domains like polyhedron domain. However, this expressiveness comes with disadvantages: First, the abstract state can have size exponential in the number of predicates. Second, the abstract domain has exponential height. The naive fixed-point computation process seems expensive especially when the final inductive invariants required to prove a given property are typically simple and small in size compared to the potential worst-case exponential representation.

⋆ This author performed the work reported here during a summer internship at Microsoft Research.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 120–135, 2009.

In this paper, we describe a technique for discovering inductive invariants over predicate abstraction that exploits the observation that the inductive invariants required for proving a given assertion typically require a small representation, instead of the worst-case exponential representation. In particular, we describe the inductive invariants using a bounded boolean structure over a given set of predicates, say DNF formulas with at most k disjuncts, where k is some small constant.[1] To achieve completeness, we can iteratively increase the value of k.

Our technique is based on the following observation: Any DNF formula with k disjuncts over a set of n predicates can be described by a truth-value assignment to $k \times n$ boolean (indicator) variables, one for each predicate p and each disjunct d denoting whether predicate p is in disjunct d. The key idea of our technique is to establish boolean constraints between the boolean indicator variables corresponding to the invariants at neighboring program locations by using the predicate cover operation[2] (The predicate cover of a formula F is the weakest formula over a set of predicates that implies F). The boolean constraint thus obtained encodes the verification condition of the program. A satisfying assignment to this boolean formula yields the inductive invariants sufficient to establish the validity of given assertions. Unsatisfiability of the boolean formula denotes that there are no inductive invariants over our choice of template structure (DNF formula with k disjuncts over the given set of predicates) to validate the assertions in the program. The size of the generated boolean formula is linear in the size of the program and the size of the predicate cover, and polynomial in the number of the predicates. Finding a satisfying assignment to the boolean formula can take exponential time in the worst-case and in theory we have still not gotten rid of the exponential factor. However, this methodology allows a direct way to leverage the engineering advances of the SAT solvers. It is noteworthy that the last decade has witnessed a revolution in SAT solvers enabling solving of industrial sized satisfiability instances.

Our constraint-based technique may offer some advantages over traditional fixed-point computation based methods. First, it does away with the iterative process of computing fixed-points, which is expensive, especially when performed on abstractions with exponential-height lattices, like predicate abstraction. Second, it cleanly splits the reasoning required of SMT formulas generated during predicate abstraction into two parts: Theory-based reasoning using predicate cover operation over small and mostly conjunctive formulas (this encodes the abstract program semantics) and SAT-based reasoning over a polynomially-sized

[1] It may appear that this observation can also be used to obtain a PTIME abstract interpretation [2] based algorithm for discovering inductive invariants with k disjuncts. However, this is not true. The domain of k-DNF formulas does not form a lattice as there is no unique LUB. The domain of formulas whose CNF representation contains at most k disjuncts in each conjunct does form a lattice of polynomial height. However, in that case, each abstract interpretation operation requires reasoning about an SMT formula in CNF form, which is NP-hard.

[2] A fundamental operation used in abstract transformers while performing abstract interpretation over predicate abstraction [1].

boolean formula[3] (this encodes the fixed-point).[4] Third, it is goal-directed and hence has the potential to be more efficient.

Our technique complements abstraction refinement techniques (such as counterexample guided abstraction refinement [4], interpolation based methods [5], forward-backward combination [6]) by equipping them with a more robust invariant generation procedure. Abstraction refinement techniques alleviate the cost involved in abstraction by iteratively refining the abstraction until an inductive invariant can be expressed. Our technique alleviates the cost of reasoning over a *given* predicate abstraction by off-loading the cost of boolean reasoning and fixed-point computation to a SAT query.

We further show how to generate *maximally-weak* preconditions using the constraint-based methodology. The key idea is to treat the precondition as an unknown relation and repeatedly search for a precondition that is weaker than the current solution until none exists. We prove that this process requires at most n satisfiability queries, where n is the number of predicates. We then describe an interesting application of maximally-weak precondition generation, namely generating *maximally-general* counterexample in case the assertions in the program are not valid.

This paper makes the following technical contributions:

- We show how to model the problem of discovering inductive invariants over predicate abstraction as the problem of finding a satisfying assignment to a boolean formula (Section 3). We also show how to extend this modeling to a context-sensitive interprocedural analysis, which is provably harder than intraprocedural analysis (Section 3.2).
- We show how to model the problem of maximally-weak precondition generation over predicate abstraction as the problem of finding satisfying assignments to (at most) n boolean formulas (Section 4). This procedure can be used to find maximally-general counterexamples to safety assertions, assuming program termination (Section 4.1).

2 Preliminaries

2.1 Program Model

We consider programs with assignments of the form $x := e$, where x denotes some variable and e denotes some expression. (Note that memory reads/writes can be modeled using this formalism by using select-update expressions.) We also

[3] It is not difficult to extend our approach to model the process of inductive invariant generation as solving only one polynomially-sized SMT query. However, the result that we present is stronger. It shows how to reduce the problem of inductive invariant generation to the problem of solving several small SMT queries over mostly conjunctive formulas and one polynomially sized SAT query.

[4] [3] presents a related technique where theory based reasoning is used to generate a boolean abstraction of the system, which is then explored using a fixed-point computation technique.

allow for assume and assert statements of the form $\texttt{assume}(p)$ and $\texttt{assert}(p)$, where p is some predicate. Since we allow for `assume` statements, without loss of generality, we assume that all conditionals in the program are non-deterministic.

2.2 Generating Verification Conditions from a Program

A *cut-set* of a program is a set of program locations (called *cut-points*) such that each cycle in the control flow graph of the program passes through some program location in the cut-set. One simple way to choose a cut-set is to include all targets of back-edges in any depth first traversal of the control-flow graph. (In case of structured programs, where all loops are natural loops, this corresponds to choosing the header node of each loop.) A *simple path* is any path that starts at a cut-point or program entry $\pi_{\texttt{entry}}$ and ends at a cut-point or program exit $\pi_{\texttt{exit}}$ without passing through any other cut-point.

We associate the program entry and exit locations as well as each cut-point π with a *relation* R^π over program variables that are live at π. The verification condition $\texttt{VC}(\tau)$ of any simple path τ between end-points π_1 and π_2 is given by the following formula:

$$\texttt{VC}(\tau) \quad = \quad R^{\pi_1} \Rightarrow \omega(\tau, R^{\pi_2})$$

The notation $\omega(\tau, R)$ denotes the weakest precondition of path τ (which is a sequence of program instructions) with respect to R and is as defined below:

$$\begin{array}{ll} \omega(x := e, R) = R[e/x] & \omega(\texttt{assume } p, R) = p \Rightarrow R \\ \omega(\tau_1; \tau_2, R) = \omega(\tau_1, \omega(\tau_2, R)) & \omega(\texttt{assert } p, R) = p \wedge R \end{array}$$

where the notation $[e/x]$ denotes substitution of x by e and may *not* be eagerly carried out across unknown relations R. Observe that the verification condition for any simple path τ between π_1 and π_2 simplifies to the following form:

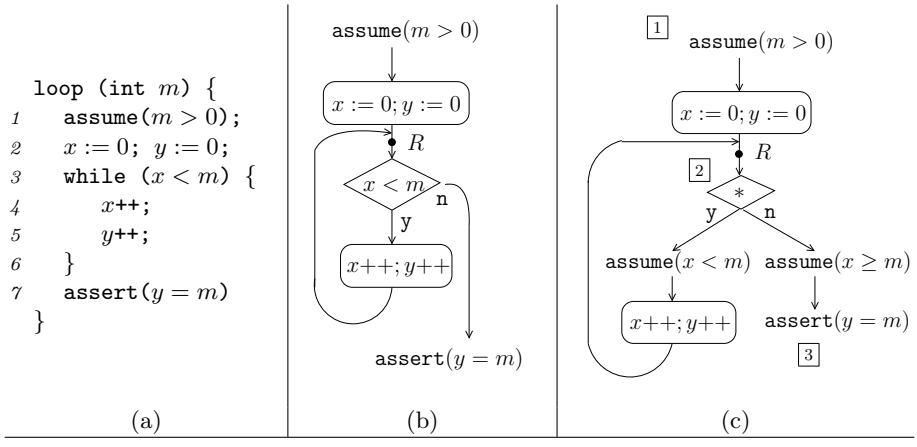
$$\texttt{VC}(\tau) \quad = \quad R^{\pi_1} \Rightarrow (G_1 \Rightarrow (G_2 \wedge R^{\pi_2}\sigma)) \tag{1}$$

where σ is some substitution, and G_1 and G_2 are known formulas obtained from the predicates that occur in assume and assert statements (on path τ), respectively, after appropriate substitutions. The following claim holds.

Claim 1. The assertions in the given program are valid iff when we set $R^{\pi_{\texttt{entry}}} = R^{\pi_{\texttt{exit}}} = \texttt{true}$ then there exist relations R^π for all cut-points π such that the verification conditions $\texttt{VC}(\tau)$ hold for every simple path τ.

3 Program Verification

Given a program with some assertions, the program verification problem is to verify whether or not the assertions are valid. The challenge in program verification is to discover the appropriate inductive invariants R^π at different program points π such that the verification conditions $\texttt{VC}(\tau)$ in Eq. 1 holds for all simple paths τ, which implies the validity of the given assertions (Claim 1). (The issue of discovering counterexamples, in case the assertions are not valid, is addressed in Section 4.1.)



(d)

$$S = \left\{ \begin{array}{ccc} x \le y, & x \ge y, & x < y, \\ x \le m, & x \ge m, & x < m \\ y \le m, & y \ge m, & y < m \end{array} \right\}$$

(e)

Fig. 1. (a) Iteration over x with an auxiliary variable y. (b) The control flow graph (CFG) with the loop invariant marked as R. (c) The CFG as modeled in our system. (d) $\texttt{VC}(\tau)$ corresponding to each simple path τ. (e) The set of predicates S.

Example. We first illustrate our constraint-based approach to invariant generation by means of a simple example. Consider the program in Figure 1(a). The program loop iterates using the loop counter x and increments an auxiliary variable y as well. Its control flow graph (CFG) is shown in Figure 1(b). The program is modeled in our system as Figure 1(c). There are three simple paths going from program entry to loop header ($\boxed{1} \to \boxed{2}$), around the loop ($\boxed{2} \to \boxed{2}$), and loop header to program exit ($\boxed{2} \to \boxed{3}$), respectively, and the verification conditions they generate (using Eq. 1) are shown in Figure 1(d). The set of predicates S over which we seek to discover our inductive invariant is shown in Figure 1(e). Let π be the program point at the loop header just *after* the join point. Suppose we make the simplifying assumption that the inductive loop relation R at π is a conjunction of some predicates from S, and we seek to discover those predicates.

The first step is to associate with each predicate $p \in S$ a boolean indicator variable b_p indicating p's presence or absence in R. Then we consider each verification condition $\texttt{VC}(\tau)$ (derived from a simple path τ using Eq. 1) in turn and generate constraints on the indicator variables:

– *Loop entry* ($\boxed{1} \to \boxed{2}$): The verification condition is $m > 0 \Rightarrow R[y \to 0, x \to 0]$, for which we generate the constraint

$$\neg b_{x<y} \wedge \neg b_{x \ge m} \wedge \neg b_{y \ge m} \qquad \text{(Ex-1)}$$

denoting that the predicates $x < y$ and $x \ge m$ and $y \ge m$ cannot be in R since they are not implied by the verification condition for loop entry.

– *Loop exit* ($\boxed{2} \rightarrow \boxed{3}$): The verification condition is $R \wedge x \geq m \Rightarrow y = m$, for which we generate the constraint

$$(b_{y \geq m} \wedge b_{y \leq m}) \vee b_{x<m} \vee (b_{x \leq y} \wedge b_{y \leq m}) \quad \text{(Ex-2)}$$

denoting that either both $y \geq m$ and $y \leq m$ belong to R, or $x < m$ belongs to R, or both $x \leq y$ and $y \leq m$ belong to R. Observe that these are the only three (maximally-weak) ways in which we can prove $y = m$ under the assumption $x \geq m$. In general, these different ways are computed by using the predicate cover operation.

– *Inductive* ($\boxed{2} \rightarrow \boxed{2}$): The verification condition is $R \wedge x < m \Rightarrow R[y \rightarrow y+1, x \rightarrow x+1]$, for which we generate the constraint

$$(b_{y \leq m} \Rightarrow (b_{y<m} \vee b_{y \leq x})) \wedge \neg b_{x<m} \wedge \neg b_{y<m} \quad \text{(Ex-3)}$$

denoting that if $y \leq m$ belongs to R, then either $y < m$ or $x \leq y \wedge y \leq x$ should also belong to R, and that the predicates $x < m$ and $y < m$ cannot be in R. The reader can easily check that this verification condition allows any other predicate p to be in R because $p \wedge x < m \Rightarrow p[y \rightarrow y+1, x \rightarrow x+1]$. These constraints are generated by considering each predicate p, finding the weakest conditions $\delta(p)$ over the set of predicates under which $p \wedge x < m \Rightarrow p[y \rightarrow y+1, x \rightarrow x+1]$ and then generating the constraint that $b_p \Rightarrow \delta(p)$. For the predicate $b_{x<m}$ and $b_{y<m}$, $\delta(p)$ is `false` and hence we generate the constraints $\neg b_{x<m}$ and $\neg b_{y<m}$. For the predicate $b_{y \leq m}$, $\delta(p)$ is $b_{y<m} \vee b_{y \leq x}$. For all other predicates, $\delta(p)$ is `true`.

Putting Eq. (Ex-1), (Ex-2), (Ex-3) together we get a SAT formula (over the boolean indicator variables) that encodes the verification condition of the program. The reader can verify that $b_{x \geq y} = b_{x \leq y} = b_{y \leq m} =$ `true` (and all others `false`) is a satisfying solution. This corresponds to R being $(x = y \wedge y \leq m)$.

3.1 Formal Constraint Generation

We now formally present our constraint-based methodology for discovering the inductive invariants R^π when they can be described using a k-DNF formula over a given set of predicates S. (We use k-DNF form for simplicity. Our methodology can also be applied to other boolean structures that are representable by a bounded number of boolean variables.) In such a case, we can represent R^π by $k \times n$ *boolean indicator variables* $b^\pi_{i,p}$ (where $1 \leq i \leq k$, $p \in S$, $n = |S|$), where the boolean variable $b^\pi_{i,p}$ denotes whether predicate p is present in the i^{th} disjunct of the invariant R^π at program point π. We show how to encode the verification condition of the program as a boolean formula ψ over the boolean indicator variables $b^\pi_{i,p}$. The boolean formula ψ is satisfiable iff there exist inductive invariants (in k-DNF form) strong enough to prove the validity of the assertions—this is the key result of the paper.

We first show how to encode the verification condition of any simple path τ as a boolean formula $\psi(\tau)$. The following three cases arise, which we consider in increasing order of difficulty:

Case 1: *Path between program entry and a cut-point.* The verification condition in Eq. 1 simplifies to the following form after substituting $R^{\pi_1} = \texttt{true}$ and expanding R^{π_2} as $\bigvee_{j=1}^{k} R_j^{\pi_2}$, where each $R_j^{\pi_2}$ is conjunction of some predicates from S.

$$G_1 \Rightarrow \left(G_2 \wedge \bigvee_{j=1}^{k} R_j^{\pi_2}\sigma \right)$$

The above constraint restricts how strong R^{π_2} can be. In particular, if $p_1 \in R_1^{\pi_2}, \ldots, p_k \in R_k^{\pi_2}$, then it must be the case that $G_1 \Rightarrow \bigvee_{j=1}^{k} p_j\sigma$. Hence, we can rewrite the above constraint as:

$$(G_1 \Rightarrow G_2) \wedge \bigwedge_{p_1,..,p_k \in S} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow (G_1 \Rightarrow \bigvee_{j=1}^{k} p_j\sigma) \right) \tag{2}$$

This can be encoded as the following boolean constraint $\psi(\tau)$ over boolean indicator variables $b_{i,p}^{\pi_2}$.

$$\psi(\tau) \quad = \quad D(G_1, G_2) \wedge \bigwedge_{p_1,..,p_k \in S} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow D(G_1, \bigvee_{j=1}^{k} p_j\sigma) \right) \tag{3}$$

where $D(A, B)$ denotes the boolean formula $\texttt{true}$ if $A \Rightarrow B$ and $\texttt{false}$ otherwise.

Case 2: *Path between a cut-point and program exit.* The verification condition in Eq. 1 simplifies to the following form after substituting $R^{\pi_2} = \texttt{true}$ and expanding R^{π_1} as $\bigvee_{j=1}^{k} R_j^{\pi_1}$, where each $R_j^{\pi_1}$ is conjunction of some predicates from S.

$$\left(\bigvee_{i=1}^{k} R_i^{\pi_1} \right) \Rightarrow (G_1 \Rightarrow G_2) \quad \text{or, equivalently,} \quad \bigwedge_{i=1}^{k} (R_i^{\pi_1} \Rightarrow (G_1 \Rightarrow G_2))$$

The above constraint restricts how weak $R_i^{\pi_1}$ can be. We can encode the above constraint as a boolean formula over the variables $b_{i,p}^{\pi}$ by considering the predicate cover[5] of $G_1 \Rightarrow G_2$. To recall, the predicate cover of a formula F over a set of predicates S, denoted by $C_S(F)$, is the weakest formula over predicates from S that implies F. Let $\phi_S(F, i, \pi)$ denote the boolean formula over boolean variables $b_{i,p}^{\pi}$ obtained after replacing each predicate p in $C_S(F)$ by $b_{i,p}^{\pi}$. The verification condition above can now be encoded as the following boolean constraint $\psi(\tau)$ over boolean indicator variables $b_{i,p}^{\pi_1}$.

$$\psi(\tau) \quad = \quad \bigwedge_{i=1}^{k} \phi_S(G_1 \Rightarrow G_2, i, \pi_1) \tag{4}$$

[5] It is a fundamental operation used in the abstract transformers while performing abstract interpretation over predicate abstraction [1].

Case 3: *Path between two adjacent cut-points.* We now combine the key ideas that we used in the above two cases to handle this more general case. The verification condition in Eq. 1 has the following form (after expanding R^{π_1} as $\bigvee_{i=1}^{k} R_i^{\pi_1}$ and R^{π_2} as $\bigvee_{j=1}^{k} R_j^{\pi_2}$, where each $R_i^{\pi_1}$ and $R_j^{\pi_2}$ is a conjunction of some predicates from S).

$$\left(\bigvee_{i=1}^{k} R_i^{\pi_1}\right) \Rightarrow \left(G_1 \Rightarrow (G_2 \wedge \bigvee_{j=1}^{k} R_j^{\pi_2}\sigma\right)$$

$$\text{or, equivalently, } \bigwedge_{i=1}^{k}\left(R_i^{\pi_1} \Rightarrow \left(G_1 \Rightarrow (G_2 \wedge \bigvee_{j=1}^{k} R_j^{\pi_2}\sigma_\tau)\right)\right) \quad (5)$$

The above constraint can be rewritten as (using the same logic used in generating the constraint in Eq. 2):

$$\bigwedge_{i=1}^{k} \bigwedge_{p_1,..,p_k \in S}\left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow \left(R_i^{\pi_1} \Rightarrow (G_1 \Rightarrow (G_2 \Rightarrow \bigvee_{j=1}^{k} p_j\sigma_\tau))\right)\right)$$

The verification condition above can be encoded as the following boolean constraint $\psi(\tau)$ over boolean indicator variables $b_{i,p}^{\pi_1}$ and $b_{i,p}^{\pi_2}$ (using the same logic used in generating the constraint in Eq. 4):

$$\psi(\tau) = \bigwedge_{i=1}^{k} \bigwedge_{p_1,..,p_k \in S}\left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow \phi_S\left((G_1 \Rightarrow (G_2 \wedge \bigvee_{j=1}^{k} p_j\sigma)), i, \pi_1\right)\right) \quad (6)$$

Observe that the constraints are generated locally from the verification condition of each simple path. Hence, the constraint based technique has the potential for efficient incremental verification (verification of a modified version of an already verified program) with support of an incremental SAT solver.

Example. The full version [7] gives examples of all the above cases over Figure 1(a).

The desired boolean formula ψ is now given by the conjunction of formulas $\psi(\tau)$ for all simple paths τ in the program. Since ψ encodes the entire verification condition of the program, it is easy to see that the following claim holds.

Claim 2. The boolean formula ψ is satisfiable iff there exist inductive invariants (in k-DNF form) strong enough to prove the validity of the given assertions.

3.2 Interprocedural Analysis

The ω computation described in Section 2.2 is applicable only in an intraprocedural setting. In this section, we show how to extend our constraint-based method to perform a precise (i.e., context-sensitive) interprocedural analysis.

Precise interprocedural analysis is challenging because the number of different calling contexts can potentially be exponential in the number of predicates over

program inputs. A standard way is to compute procedure summaries, which are relations between procedure inputs and outputs. These summaries are usually structured as sets of pre/postcondition pairs (A_i, B_i), where A_i is some relation over procedure inputs and B_i is some relation over procedure inputs and outputs. A pair (A_i, B_i) denotes that whenever the procedure is called in a context that satisfies A_i, the procedure ensures that the outputs satisfy the constraint B_i. However, the efficient construction of relevant pre/postcondition pairs is unclear.

In this section, we show that the constraint-based approach is particularly suited to discovering useful pre/postcondition (A_i, B_i) pairs. The key idea is to observe that the desired behavior of most procedures can be captured by a small number of such (unknown) pre/postcondition pairs. We then replace the procedure calls by these unknown behaviors and assert that the procedure, in fact, has such behaviors in an assume-guarantee style reasoning. Our encoding requires the summary to be only as precise as is required for verification of the given assertions.

Procedure bodies: Without loss of generality, let us assume that a procedure does not read/modify any global variables; instead all global variables that are read by the procedure are passed in as inputs, and all global variables that are modified by the procedure are returned as outputs. Suppose there are q interesting calling contexts for the procedure $P(\boldsymbol{x})\{S; \texttt{return } \boldsymbol{y};\}$ with the vector of formal arguments $\boldsymbol{x}$ and vector of return values $\boldsymbol{y}$. (In practice, the value of q can be iteratively increased until the constraint system is satisfiable.) We can summarize the behavior of procedure P for these q interesting calling contexts using q tuples (A_i, B_i) for $1 \leq i \leq q$, where A_i is some (unknown) relation over $\boldsymbol{x}$, and B_i is some (unknown) relation over $\boldsymbol{x}$ and $\boldsymbol{y}$. We ensure this by generating constraints for each i as below:

$$\texttt{assume}(A_i);\ S;\ \texttt{assert}(B_i) \tag{7}$$

Procedure calls: For simplicity, we assume that the cut-set includes all program locations before any procedure call. For any simple path τ that starts with a procedure call $\boldsymbol{v} := P(\boldsymbol{u})$, let τ_i denote the simple path in which the procedure call is replaced by the following code fragment, where $\boldsymbol{t}$ is a fresh set of variables.

$$\texttt{assert}(A_i[\boldsymbol{u}/\boldsymbol{x}]); \texttt{assume}(B_i[\boldsymbol{u}/\boldsymbol{x}, \boldsymbol{t}/\boldsymbol{y}]); \boldsymbol{v} := \boldsymbol{t}; \tag{8}$$

The boolean formula $\psi(\tau_i)$ that encodes the verification condition of the simple path τ_i can be computed using the method described in Section 3. The formula that encodes the verification condition corresponding to τ is $\psi(\tau) = \bigvee_{i=1}^{q} \psi(\tau_i)$.

Example. In the full version [7] we illustrate the technique over some examples.

4 Maximally-Weak Precondition Inference

Given a program with some assertions, the problem of weakest precondition generation is to infer the weakest precondition $R^{\pi_{\texttt{entry}}}$ such that whenever the program is run in a state that satisfies $R^{\pi_{\texttt{entry}}}$, the assertions in the program hold. This weakest precondition inference problem is harder than program verification, and relatively

few techniques exist for it. Since a precise solution is undecidable, we work with a relaxed notion of weakest precondition. For a given template structure, we say that A is a *maximally-weak* precondition if A is a precondition that fits the template and there does not exist a weaker precondition than A with similar properties.

In this section, we present a constraint-based approach to inferring maximally-weak preconditions under a given template. Specifically, we show how to generate a conjunctive maximally-weak precondition for a given program with assertions. A k-DNF maximally-weak precondition can then be obtained by taking disjunctions of k disjoint conjunctive maximally-weak preconditions, generated iteratively. Our constraint-based approach permits an elegant maximally-weak precondition inference technique based on the monotonicity of implication for CNF formulae over a given set of predicates.

The first step is to treat the precondition $R^{\pi_{\texttt{entry}}}$ as an unknown relation in Eq. 1, unlike in program verification where we set $R^{\pi_{\texttt{entry}}}$ to be `true`. However, this small change merely encodes that any consistent assignment to $R^{\pi_{\texttt{entry}}}$ is a valid precondition, not necessarily maximally-weak. In fact, when we run our tool with this small change, it returns `false` as a solution for $R^{\pi_{\texttt{entry}}}$. Note that `false` is always a valid precondition, but not necessarily maximally-weak.

We use an iterative approach to generating a conjunctive maximally-weak precondition as follows. We add the constraint that the precondition $R^{\pi_{\texttt{entry}}}$ should be weaker than the current solution T (which is initialized to `false`) to the verification condition (in Eq. 1). This constraint is encoded by the boolean formula $(\gamma_1 \wedge \neg\gamma_2)$, where γ_1 and γ_2 are boolean formulae over the boolean variables $b_p^{\pi_{\texttt{entry}}}$ that encode the constraints $T \Rightarrow R^{\pi_{\texttt{entry}}}$ and $R^{\pi_{\texttt{entry}}} \Rightarrow T$, respectively and are computed using the technique described in Section 3.

Once a maximally-weak conjunctive precondition has been found, we repeat the process to generate other maximally-weak conjunctive preconditions. In order to ensure that we get a precondition that is disjoint from maximally-weak preconditions already found, we add an additional constraint $\neg\gamma_3$, where γ_3 is the boolean formula over the boolean variables $b_p^{\pi_{\texttt{entry}}}$ that encodes the constraint $R^{\pi_{\texttt{entry}}} \Rightarrow \bigvee_i T_i$, where T_i are the conjunctive maximally-weak preconditions that have already been discovered.

Example. We again consider Figure 1(a) but with line 1 ($x := 0; y := 0$) removed and infer maximally-weak preconditions between x, y, m using the predicate set S shown in Figure 1(e). We generate two conjunctive maximally-weak preconditions: $(y = m \wedge x \geq m)$ and $(x = y \wedge x < m)$; their disjunction yields the disjunctive maximally-weak precondition.

4.1 Maximally-General Counterexample Inference

Since program analysis is an undecidable problem, we cannot have tools that can prove correctness of all correct programs or find bugs in all incorrect programs. Hence, to maximize the practical success rate of verification tools, it is desirable to search for both proofs of correctness as well as counterexamples in parallel. Earlier, we showed how to find proofs of correctness of given assertions. In this

section, we show how to find *maximally-general* counterexamples to given assertions (under a given template structure).

The problem of generating a maximally-general counterexample for a given set of safety assertions involves finding a maximally-general characterization of inputs that leads to violation of some reachable safety assertion. Generating a maximally-general counterexample is more desirable than generating a concrete counterexample, and can aid in, say, program debugging. For example, it is more useful to know that there is an assertion failure whenever $x < y$ as opposed to knowing that there is an assertion failure when $x = 0 \wedge y = 3$.

We show next how to find a maximally-general counterexample using the techniques discussed in Section 4 under the assumption that the given program is terminating, i.e., the program exit is always reached. The basic idea is to reduce the problem to that of finding a maximally-weak precondition for some safety property. This reduction involves constructing another program from the given program `Prog` using the following transformation, $T_{\texttt{err}}(\texttt{Prog})$: We introduce a new variable `error` that is set to 0 at the beginning of the program. Whenever violation of the given safety property occurs (i.e., the negation of any of the safety assertions holds), the variable `error` is set to 1 and the control jumps to the end of the program. We assert that `error` $= 1$ at the end of the program, and remove the original safety assertions from the program.

Claim 3. Let `Prog` be a terminating program with some safety assertions. Then, `Prog` has an assertion violation iff the assertions in program $T_{\texttt{err}}(\texttt{Prog})$ hold.

The significance of Claim 3 is that now we can use maximally-weak precondition inference (Section 4) on the transformed program to discover maximally-general characterization of inputs under which there is a safety violation in the original program. We need to track the new boolean variable `error` in the transformed program and therefore add `error` $= 1$ and `error` $= 0$ to the predicate set.

Example. Consider the program shown in Figure 2(a), which we instrument with the error variable to obtain the program in Figure 2(b). Our maximally-weak precondition inference generates $(x < m \wedge y \geq x)$ as the maximally-general

```
  err (int m) {
1   while (x < m) {
2       x++;
3       y++;
4       assert(y < m)
5   }
  }
```

(a)

```
  err (int m) {
1   error := 0;
2   while (x < m) {
3       x++; y++;
4       if (y ≥ m)
5           error := 1; goto L;
6   }
7   L: assert(error = 1)
  }
```

(b)

Fig. 2. (a) Example with safety assertion $y < m$. (b) Instrumented program. We compute $(x<m \wedge y \geq x)$ as a maximally-general counterexample that violates the assertion.

counterexample that violates the assertion $y < m$. Note that we need k to be at least 2 since the inductive invariant (at the loop header) for establishing the counterexample is $(x < m \wedge y \geq x) \vee (\texttt{error} = 1 \wedge y \geq x)$).

Observe the importance of introducing the error variable. An alternative that one might consider is to simply negate the original safety assertion instead of introducing an error variable. This is incorrect for two reasons: (a) It is too stringent a criterion because it insists that in each iteration of the loop the original assertion does not hold, (b) It does not ensure reachability and allows for those preconditions under which the assert statement is never executed. In fact, running our tool with such an alternative transformation yields two conjunctive maximally-weak preconditions—$(x \geq m)$ and $(x < m \wedge y \geq m - 1)$ of which the former does not describe a counterexample, while the latter does not describe a conjunctive maximally-general counter-example.

5 Experiments

In this section we demonstrate the viability of a constraint-based approach by uniformly discovering invariants for programs for which specialized techniques [8,9,10] have been proposed.

The results of invariant generation for program verification are shown in Table 1(a). The first set of columns indicate the programs[6], the parameters (number of disjuncts k, and size of predicate set n), and the number of variables and clauses in the CNF formula. The second set indicates the time (in seconds) on generating the program constraints (CG), generating the CNF formula (CNF) and solving the resulting SAT instance (SAT). We use Z3 [11] as our SAT solver.

The first set of examples require conjunctive invariants ($k = 1$). The first program (*counter*) is a loop iteration with a counter from $1 \ldots m$. The second (*lockstep*, shown in Figure 1) is also a counter iteration but with another variable counting in lock-step. The third (*nested*) consists of two nested counter loops. The next program (*twoloop*) consists of two counter loops one after the other. The last two examples need two invariants, one at each loop header.

Table 1. Results

Program	Number of				Time for		
	n	k	vars	clauses	CG	CNF	SAT
counter	12	1	12	21	0.23	0.14	0.04
ex1a [8]	12	1	12	22	0.23	0.15	0.04
lockstep	5	1	5	8	0.23	0.11	0.03
nested	16	1	32	62	0.23	0.26	0.04
twoloop	20	1	40	79	0.23	0.36	0.04
ex2 [9]	12	2	24	72	0.23	0.14	0.04
ex1b [8]	20	2	40	1704	0.23	10.68	0.06
ex3 [10]	20	2	40	1782	0.23	8.53	0.06

(a) Program Verification

Program	Number of					Time for		
	n	k	vars	clauses	sol	CG	CNF	SAT
counter	12	1	24	1345	1	0.23	0.44	0.05
ex1a [8]	14	1	28	1857	2	0.25	0.67	0.07
lockstep	9	1	18	584	2	0.23	0.29	0.05
nested	18	1	54	2866	2	0.23	1.52	0.09
twoloop	20	1	60	3778	3	0.23	1.86	0.16
ex2 [9]	12	2	36	4588	1	0.23	1.16	0.11
ex1b [8]	20	2	60	7548	1	0.23	14.22	0.09
ex3 [10]	13	2	39	2031	4	0.23	3.90	0.14

(b) Precondition Inference

[6] Available at http://research.microsoft.com/users/sumitg/benchmarks/pa.html

The second set requires disjunctive invariants ($k = 2$) and are from recent work on sophisticated invariant generation techniques like CFG elaboration (*ex2* [9]), probabilistic inference (*ex3* [10]), and sophisticated widening (*ex1b* [8]).

Our technique uniformly discovers invariants over predicate abstraction for all these examples. Our base predicates are difference constraints over the program variables with small constants. Program parsing and constraint generation takes 0.23s. Our preliminary tool uses an unoptimized implementation of predicate cover and therefore spends most of its time in CNF generation, which can be improved easily. Solving the resulting CNF constraints takes 0.04s on average. Our preliminary tool also shows a noticeable overhead when a disjunctive invariant at the loop header causes case enumeration during CNF generation (in *ex3* [10] and *ex1b* [8]). However, even for large SAT instances in these cases, solutions are generated by the solver in very reasonable time, demonstrating the viability of a constraint-based approach.

5.1 Maximally-Weak Precondition Inference

The SAT solver that we used tends to generate a *maximally-false* satisfying assignment to a satisfiable boolean formula, as a result of which we obtained conjunctive maximally-weak preconditions in the first query and did not have to iterate n times. A satisfying assignment A to a boolean formula is *maximally-false* if by changing the truth values of any of the boolean variables from `true` to `false` in assignment A transforms A to an unsatisfying assignment.

We exploit this property by adding an additional constraint to the system, which in practice improves performance. The added clauses constrains the maximally-weak precondition to be saturated, i.e., for all predicates p_1, p_2, p_3, if $p_1 \wedge p_2 \Rightarrow p_3$, we add the constraint:

$$b_{p_1}^{\pi_{\texttt{entry}}} \wedge b_{p_2}^{\pi_{\texttt{entry}}} \Rightarrow b_{p_3}^{\pi_{\texttt{entry}}} \tag{9}$$

Claim 4. A *maximally-false* satisfying assignment to the boolean formula ψ (that encodes the verification condition of the program) along with the constraint in Eq. 9 yields a conjunctive maximally-weak precondition.

The results for precondition inference are shown in Table 1(b). Our tool generates all maximally-weak preconditions when multiple incomparable ones exist. Therefore, in addition to the number of predicates n, disjuncts k, variables and clauses in the CNF, we also report the number of solutions generated. In such cases, we report the cumulative time required for generating all solutions.

Due to the tendency of the SAT-solver to generate a maximally-false assignment, our tool produced valid maximally-weak preconditions in the first iteration for all but two programs, *ex2* [9] and *twoloop*, which required two iterations.

6 Related Work

Constraint-based techniques have been recently used for discovering linear arithmetic invariants (conjunctive invariants [12,13,14,15] as well as disjunctive invariants [16] in the context of verifying safety properties as well as discovering ranking

functions for proving termination [17,18]). Constraint-based techniques have also been applied for discovering non-linear polynomial invariants [14,19] and invariants in the combined theory of linear arithmetic and uninterpreted functions [20]. In contrast, this paper extends the applicability of constraint-based methodology to the important domain of predicate abstraction, where the predicates can be from any theory. There are two key technical differences between the earlier work that focused on arithmetic invariants and the current work based on predicate abstraction: (a) The key principle behind a constraint-based methodology is to convert universal quantification into existential quantification in the verification condition. In this respect, the earlier work uses Farkas' lemma, while the current work uses the predicate cover operation. (b) The earlier work translates the problem of discovering arithmetic invariants into solving polynomial constraints[7], while in contrast the proposed technique translates the problem of discovering invariants over a given set of predicates into solving a SAT constraint. The latter is more desirable since we have good off-the-shelf SAT solvers.

Constraint-based techniques, being goal-directed, work naturally in program verification mode where the task is to discover inductive loop invariants for verification of given assertions. As a result, earlier work on constraint-based techniques (with the exception of [16]) has been limited to program verification as opposed to other program analysis problems such as precondition generation. This paper demonstrates the applicability of constraint-based methodology to the problem of maximally-weak precondition generation, which in turn can be used for generation of maximally-general counterexamples (assuming program termination). The technique used for precondition generation in [16] is based on encoding *locally-pointwise weakest* property for linear arithmetic constraints, while the technique used for precondition generation in this paper relies on monotonicity of implication of a conjunctive set of predicates.

SATURN [21] also uses SAT-solving, but for bug-finding in loop-free programs. (Programs with loops are modeled by unrolling loops.) Theoretically, it is well known that loop-free programs can be modeled as Boolean circuits. SATURN's contribution is primarily engineering-based; it illustrates that the SAT queries that are generated from real programs with complicated constructs can be efficiently solved in practice. In contrast, we focus on invariant inference for correctness proofs and show how programs with loops can be abstracted as Boolean circuits. Additionally, our work finds *maximally-general* bugs in programs with loops.

7 Conclusions and Future Work

We present a constraint-based technique for discovering inductive program invariants over predicate abstraction. Our technique pushes the burden of fixed-

[7] [16] further proposes solving the quadratic inequalities using bit-vector modeling, thus effectively translating into SAT constraints; however, this reduction to SAT is artificial in that it is used only to approximate the SMT query, because current solvers cannot generate models for SMT queries that involve multiplication.

point computation and boolean reasoning to a SAT-solver by encoding program verification conditions as SAT-constraints over boolean indicator variables. A solution to the SAT instance maps directly to inductive program invariants that prove the validity of given program assertions. We lift the verification procedure to interprocedural setting and additionally infer maximally-weak preconditions that can be used for maximally-general bug finding. We present encouraging preliminary results using a prototype implementation. Experimentation with large programs and comparison with alternative techniques remains future work.

References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by constr. or approx. of fixpoints. In: POPL, pp. 238–252 (1977)
3. Colón, M., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 293–304. Springer, Heidelberg (1998)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
5. McMillan, K.L.: Appl. of craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
6. Cousot, P., Ganty, P., Raskin, J.F.: Fixpoint-guided abstraction refinements. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 333–348. Springer, Heidelberg (2007)
7. Technical Report MSR-TR-2008-163, Microsoft Research (2008)
8. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 474–488. Springer, Heidelberg (2006)
9. Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006)
10. Gulwani, S., Jojic, N.: Program verification as prob. inference. In: POPL (2007)
11. de Moura, L.M., Bjørner, N.: Eff. E-Matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
12. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
13. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)
14. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: POPL, pp. 318–329 (2004)
15. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)

16. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, pp. 281–292 (2008)
17. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
18. Bradley, A.R., Manna, Z., Sipma, H.B.: Lin. ranking with reach. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
19. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Deduction and Applications (2005)
20. 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)
21. Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug det. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 139–143. Springer, Heidelberg (2005)

Reducing Behavioural to Structural Properties of Programs with Procedures

Dilian Gurov[1,⋆] and Marieke Huisman[2,⋆⋆]

[1] Royal Institute of Technology, Stockholm, Sweden
[2] University of Twente, Netherlands

Abstract. There is an intimate link between program structure and behaviour. Exploiting this link to phrase program correctness problems in terms of the structural properties of a program graph rather than in terms of its unfoldings is a useful strategy for making analyses more tractable. This paper presents a characterisation of behavioural program properties through sets of structural properties by means of a translation. The characterisation is given in the context of a program model based on control flow graphs of sequential programs with possibly recursive procedures, and properties expressed in a fragment of the modal μ-calculus with boxes and greatest fixed-points only. The property translation is based on a tableau construction that conceptually amounts to symbolic execution of the behavioural formula, collecting structural constraints along the way. By keeping track of the subformulae that have been examined, recursion in the structural constraints can be identified and captured by fixed-point formulae. The tableau construction terminates, and the characterisation is exact, *i.e.,* the translation is sound and complete. A prototype implementation has been developed. We discuss several applications of the characterisation, in particular compositional verification for behavioural properties, based on maximal models.

1 Introduction

The relationship between a program's syntactical structure and its behaviour is fundamental in program analysis. For example, type systems analyse the structure of a program to deduce properties about its behaviour, while program synthesis studies how to realise a program structure for a desired program behaviour. The relationship is often exploited to phrase program correctness problems in terms of the structure of a program rather than in terms of its behaviour, in order to make analyses more tractable. If program data is abstracted away, and only the *control flow* of programs with (possibly recursive) procedures is considered, the relation between structure and behaviour is well-understood in one

⋆ Partially funded by the IST FP6 programme of the EC, under the IST-FP6-STREP-27004 S3MS project.

⋆⋆ Work done while at INRIA Sophia Antipolis. Partially funded by the IST FET programme of the EC, under the IST-2005-015905 MOBIUS project.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 136–150, 2009.

direction: program structure, essentially a finite "program graph", can be represented by a pushdown system that induces program behaviour as an "unfolding" of the structure in a context-free manner. This representation has been exploited widely, for example for interprocedural dataflow analysis (*e.g.,* in [17]) and for model checking of behavioural properties (*e.g.,* in [8]). However, in the other direction, this relationship is much less understood: given a program behaviour, how can one capture the program structures that admit this behaviour?

Both program structure and behaviour can be specified by *temporal logic* formulae: structural properties are concerned with the textual sequencing of instructions in a program, while behavioural properties consider their executional sequencing. The relationship between structure and behaviour is naturally expressed at the logic level through the following two questions:

(1) when does a structural property entail a behavioural one and,
(2) can a behavioural property be characterised by a finite set of structural ones?

This extended abstract (the accompanying report [11] contains proofs and more examples) addresses this *characterisation problem* in the context of a program model based on control flow graphs of sequential programs with procedures (*i.e.*, program data is abstracted away), for properties expressed in a fragment of the modal μ-calculus with boxes and greatest fixed-points only. This temporal logic is suitable for expressing safety properties (*cf.* [4]) in terms of sequences of method invocations, such as security policies restricting access to given resources by means of API method calls (*cf.* [18]). In previous work [12], we showed how this logic can be used for the specification and compositional verification of safety properties, both on the structural and on the behavioural level, and provided tool support and case studies. In particular, we derived an algorithmic solution to problem (1) stated above (see [12, p. 855]). Here, we give a precise solution to the (more complex) problem (2), showing that every disjunction-free behavioural formula can be characterised by a finite set of structural formulae: a program satisfies the behavioural formula if and only if it satisfies *some* structural formula from the set. For example, the results of this paper allow to derive that the behavioural property "method a never calls method b" is characterised by the (singleton set) structural property "in (the text of) method a, every call–to–b instruction is preceded by some call–to–a instruction" (and hence, due to recursion, control never reaches a call–to–b instruction).

Our solution is constructive, by means of a translation Π from behavioural properties into sets of structural properties. The translation has been implemented in Ocaml and can be tested online [10]. It conceptually amounts to a symbolic execution of the behavioural formula, collecting induced structural constraints along the way. A considerable difficulty is presented by (greatest fixed-point) recursion in the behavioural formula, which has to be captured by recursion in the structural ones (in the absence of recursion it is considerably easier to define such a translation, as we show in [14]). We handle recursion by means of a *tableau construction* that maintains (during the symbolic execution) a symbolic "call stack" indicating which subformulae have been explored for which method. We use this stack to (1) identify when a (sub)formula has

been sufficiently explored, so that a branch of the tableau can be finished, and (2) to identify recursion in the collected structural constraints and capture this by fixed-point formulae. We prove that the construction terminates. Moreover, we show that the construction is *sound*, and in case the behavioural formula is disjunction-free, also *complete,* by viewing the tableau system as a proof system.

Applications. In addition to its foundational value, the characterisation is useful in various ways. In earlier work, we defined a *maximal model* construction for the logic considered here, and adapted it to the construction of maximal program structures from structural properties [12]. The combination of this construction with the property translation Π provides a solution to the problem of computing maximal program structures from behavioural properties. As Section 4 shows, this can be exploited to extend the *compositional verification* technique of [12], where local assumptions are required to be structural, to local behavioural properties. Further, the translation can be used to reduce infinite-state verification of behavioural control flow properties to finite-state verification of structural properties. Thus, tools supporting structural properties only can in effect be used for verifying behavioural properties. Moreover, in a mobile code deployment scheme, where the security policies of the platform are given as behavioural control flow properties, translating these into structural properties of the loaded applications enables efficient on-device conformance checking via static analysis.

Related Work. Our property translation has been motivated by our previous work on adapting Grumberg and Long's approach of using maximal models for compositional verification [9] in the context of control flow properties of sequential programs with procedures [12]. Maximal models can also be constructed for the full μ-calculus, but require representations beyond ordinary labelled transition systems, such as the focused transition systems proposed by Dams and Namjoshi [7]. Our research is also related to previous work on tableau systems for the verification of infinite-state systems [6,19], model checking based on pushdown systems [5,8] or recursive state machines [2], temporal logics for nested calls and returns [1,3], interprocedural dataflow analysis [17], and abstract interpretation (*cf. e.g.,* the completeness result of [16]). However, these analyses infer from the structure of a given program facts about its behaviour; in contrast, our analysis infers, for *all* programs satisfying a certain behaviour, facts about the structure from facts about that behaviour.

Organisation. Section 2 formally defines the program model and logic. Next, Section 3 defines the translation, by means of the tableau construction. Section 4 uses the characterisation to develop a sound and complete compositional verification principle for local behavioural properties, while Section 5 concludes with a discussion of possible extensions and optimisations.

2 Preliminaries: Program Model and Logic

This section summarises the definitions of program model, logic and satisfaction. These are first defined generally, and then instantiated at structural and behavioural level. We refer the reader to [12] for more details.

2.1 Specification and Logic

First, we define the general notions of model and specification.

Definition 1. (Model, Specification) *A* model *is a (Kripke) structure* $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$*, where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labelled transition relation,* A *a set of atomic propositions, and* $\lambda\colon S \rightarrow \mathcal{P}(A)$ *a valuation, assigning to each state* s *the set of atomic propositions that hold in* s*. A* specification $\mathcal{S}$ *is a pair* $(\mathcal{M}, E)$*, with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of entry states.*

As property specification language, we use the fragment of the modal μ-calculus [15] with boxes and greatest fixed-points only. This fragment is suitable for expressing safety properties and is capable of characterising simulation (*cf.* [12]). Throughout, we fix a set of labels L, a set of atomic propositions A, and a set of propositional variables V.

Definition 2. (Logic) *The formulae of our logic are inductively defined by:* $\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\,\phi \mid \nu X.\phi$*, where* $p \in A$*,* $a \in L$ *and* $X \in V$*.*

Satisfaction on states $(\mathcal{M}, s) \models \phi$ (also denoted $s \models^{\mathcal{M}} \phi$) is defined in the standard fashion [15]. For instance, formula $[a]\,\phi$ holds of state s in model $\mathcal{M}$ if ϕ holds in all states accessible from s via a transition labelled a. A specification $(\mathcal{M}, E)$ satisfies a formula if all its entry states E satisfy the formula. The constant formulae *true* (denoted tt) and *false* (ff) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$. In our translation of simulation logic formulae we allow sequences α of labels to appear in box modalities, with the obvious translation $\widehat{\cdot}$ to standard formulae: $\widehat{[\epsilon]\,\psi} = \psi$ and $\widehat{[l \cdot \alpha]\,\psi} = [l]\,\widehat{[\alpha]\,\psi}$, where ϵ denotes the empty sequence, and ψ is already a standard formula.

2.2 Control Flow Structure and Behaviour

Our program model is control–flow based and thus over–approximates actual program behaviour. This approach is sound, since we focus on safety properties. We define two different views on programs: a structural and a behavioural view. Both views are instantiations of the general notions of model and specification. Notice in particular that these instantiations yield a structural and a behavioural version of the logic.

Control Flow Structure. As we abstract away from all data, program structure is defined as a collection of control flow graphs (or flow graphs), one for each of the program's methods. Let *Meth* be a countably infinite set of method names. A method specification is an instance of the general notion of specification.

Definition 3. (Method specification) *A* flow graph *for* $m \in Meth$ *over a set* $M \subseteq Meth$ *of method names is a finite model* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$*, with* V_m *the set of* control nodes *of* m*,* $L_m = M \cup \{\varepsilon\}$*,* $A_m = \{m, r\}$*, and* $\lambda_m\colon V_m \rightarrow \mathcal{P}(A_m)$ *so that* $m \in \lambda_m(v)$ *for all* $v \in V_m$ *(*i.e.*, each node is tagged*

```
class Number {
   public static boolean even(int n){
      if (n == 0)
         return true;
      else
         return odd(n-1);
   }

   public static boolean odd(int n){
      if (n == 0)
         return false;
      else
         return even(n-1);

   }}
```

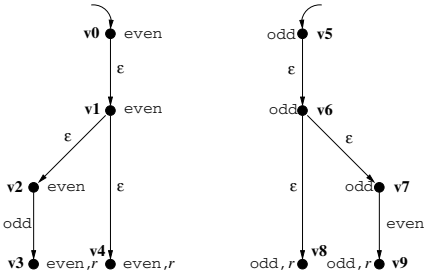


Fig. 1. A simple Java class and its flow graph

with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are return points. *A* method specification *for $m \in Meth$ over M is a pair $(\mathcal{M}_m, E_m)$, s.t. $\mathcal{M}_m$ is a flow graph for m over M and $E_m \subseteq V_m$ a non–empty set of* entry points *of m.*

Next, we define flow graph interfaces. These ensure that control flow graphs can only be composed if their interfaces match.

Definition 4. (Flow graph interface) *A* flow graph interface *is a pair $I = (I^+, I^-)$, where $I^+, I^- \subseteq Meth$ are finite sets of names of* provided *and* required *methods, respectively. The* composition *of two interfaces $I_1 = (I_1^+, I_1^-)$ and $I_2 = (I_2^+, I_2^-)$ is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$. An interface $I = (I^+, I^-)$ is* closed *if $I^- \subseteq I^+$.*

The flow graph of a program is essentially the (disjoint) union of its method graphs. To formally define the notion *flow graph with interface*, we use the notion of disjoint union of specifications $\mathcal{S}_1 \uplus \mathcal{S}_2$, where each state is tagged with 1 or 2, respectively, and $(s, i) \xrightarrow{a}_{\mathcal{S}_1 \uplus \mathcal{S}_2} (t, i)$, for $i \in \{1, 2\}$, if and only if $s \xrightarrow{a}_{\mathcal{S}_i} t$.

Definition 5. (Flow graph with interface) *A* flow graph $\mathcal{G}$ with interface I, *written $\mathcal{G} : I$, is defined inductively by*

- $(\mathcal{M}_m, E_m) : (\{m\}, M)$ *if $(\mathcal{M}_m, E_m)$ is a method specification for $m \in Meth$ over M, and*
- $\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ *if $\mathcal{G}_1 : I_1$ and $\mathcal{G}_2 : I_2$.*

A flow graph is closed if its interface is closed (*i.e.,* it does not require any external methods), and is *clean* if return points have no outgoing edges. In the sequel, we shall assume, without loss of generality, that flow graphs are clean. Satisfaction, instantiated to flow graphs, is called structural satisfaction $\models_s$.

Example 1. Figure 1 shows a Java class and its (simplified) flow graph with interface $(\{\texttt{even}, \texttt{odd}\}, \{\texttt{even}, \texttt{odd}\})$. This contains two method specifications, for method `even` and for method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source. For this flow graph, the structural formula $\nu X.\ [\texttt{even}]\, r \wedge [\texttt{odd}]\, r \wedge [\varepsilon]\, X$ expresses the property that "on every path from a program entry node, the first encountered call edge leads to a return node", in effect specifying that the program is tail-recursive.

Control Flow Behaviour. Next, we instantiate specifications on the behavioural level. We use transition label τ to designate internal transfer of control, label $m_1 \,\mathsf{call}\, m_2$ to designate an invocation of method m_2 by method m_1, and label $m_2 \,\mathsf{ret}\, m_1$ for a corresponding return from the call.

Definition 6. (Behaviour) *Let $\mathcal{G} = (\mathcal{M}, E) : I$ be a closed flow graph where $\mathcal{M} = (V, L, \to, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is defined as model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$,* i.e., *states (or* configurations*) are pairs of control points v and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\to_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

$$
\begin{array}{rll}
[\text{transfer}] & (v,\sigma) \xrightarrow{\tau}_b (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \\
[\text{call}] & (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_b (v_2, v_1' \cdot \sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1',\ v_1 \models \neg r, \\
 & & \quad v_2 \models m_2,\ v_2 \in E \\
[\text{return}] & (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_b (v_1,\sigma) & \text{if } m_1, m_2 \in I^+,\ v_2 \models m_2 \wedge r,\ v_1 \models m_1
\end{array}
$$

The set of initial states is defined by $E_b = E \times \{\epsilon\}$.

Flow graph behaviour can alternatively be defined in terms of *pushdown automata* (PDA) [12, Def. 34]. This can be exploited by using PDA model checking for verifying behavioural properties (see for instance [5,8]).

Also on the behavioural level, we instantiate the definition of satisfaction: we define $\mathcal{G} \models_b \phi$ as $b(\mathcal{G}) \models \phi$. The resulting behavioural logic is sufficiently powerful to express the class of *security policies* that can be defined by means of finite-state security automata (*cf. e.g.* [18]).

Example 2. Consider the flow graph from Example 1. Because of possible unbounded recursion, it induces an infinite-state behaviour. One example run (*i.e.*, linear execution) through this behaviour is represented by the following path from an initial to a final configuration:

$$
\begin{array}{l}
(v_0,\epsilon) \xrightarrow{\tau}_b (v_1,\epsilon) \xrightarrow{\tau}_b (v_2,\epsilon) \xrightarrow{\text{even call odd}}_b (v_5, v_3) \xrightarrow{\tau}_b (v_6, v_3) \xrightarrow{\tau}_b (v_7, v_3) \xrightarrow{\text{odd call even}}_b \\
(v_0, v_9 \cdot v_3) \xrightarrow{\tau}_b (v_1, v_9 \cdot v_3) \xrightarrow{\tau}_b (v_4, v_9 \cdot v_3) \xrightarrow{\text{even ret odd}}_b (v_9, v_3) \xrightarrow{\text{odd ret even}}_b (v_3, \epsilon)
\end{array}
$$

For this flow graph, the behavioural formula $\texttt{even} \Rightarrow \nu X.\ [\texttt{even}\,\mathsf{call}\,\texttt{even}]\,\mathsf{ff} \wedge [\tau]\,X$ expresses the property "in every program execution that starts in method `even`, the first call is not to method `even` itself".

More example properties and realistic program specifications can be found in [12].

3 Mapping Behavioural into Structural Properties

This section defines a mapping Π from interfaces and behavioural properties to sets of structural properties. As mentioned above, the implementation of the mapping can be tested online. Throughout the section we assume that behavioural properties are disjunction-free; in Section 5 we discuss how Π can be

extended to behavioural formulae with disjunction, though at the expense of completeness. We show that Π computes, from a behavioural property ϕ and closed interface I, a set of structural formulae that characterises ϕ and I. That is, for any (closed) flow graph $\mathcal{G}$ with interface I and any behavioural formula ϕ that only mentions labels that are in the behaviour of $\mathcal{G}$ (*i.e.*, L_b in Definition 6):

$$\mathcal{G} \models_b \phi \;\Leftrightarrow\; \exists \chi \in \Pi_I(\phi).\, \mathcal{G} \models_s \chi \tag{1}$$

To deal with the fixed-point formulae of the logic, mapping Π is defined with the help of a *tableau construction*. A behavioural formula ϕ gives rise to a (maximal) tableau that induces a set of structural formulae through its leaves. The constructed tableau is finite, *i.e.,* tableau construction terminates.

3.1 Tableau Construction

Our translation is based on a *symbolic execution* of the behavioural property by means of a *tableau construction*. When tracing a symbolic execution path, we tag all subformulae of the formula with unique propositional constants from a set $\mathcal{C}onst$. We use a global map $\mathcal{S} : \phi \to \mathcal{C}onst$ to map formulae to their tags. We consider $\mathcal{S}$ as an implicit parameter of the tableau construction. The tableau construction operates on sequents of the shape $\vdash_{H,U,C} \phi$ parametrised on:

- a non-empty *history stack* $H \in (I^+ \times (I^- \cup \{\varepsilon\} \cup \mathcal{C}onst)^*)^+$, where each element is a pair (i, F) consisting of a method name $i \in I^+$ (called the current method) and a sequence $F \in (I^- \cup \{\varepsilon\} \cup \mathcal{C}onst)^*$ of edge labels and propositional constants abbreviating subformulae of ϕ (called frame). For any frame F, we use $\widetilde{F}$ to denote this frame cleaned from propositional constants $X \in \mathcal{C}onst$:

$$\widetilde{\epsilon} = \epsilon \qquad \widetilde{m \cdot \sigma} = m \cdot \widetilde{\sigma} \qquad \widetilde{\varepsilon \cdot \sigma} = \varepsilon \cdot \widetilde{\sigma} \qquad \widetilde{X \cdot \sigma} = \widetilde{\sigma}$$

- a *fixed-point stack* U, defining an environment for propositional variables by means of a sequence of definitions of the shape $X = \nu X.\psi$. An *open* formula ϕ in a sequent parametrised by U can then be understood via a suitable notion of substitution, based on the standard notion of substitution $\psi\{\theta/X\}$ of a formula θ for a propositional variable X in a formula ψ: the *substitution* of ϕ under U is inductively defined as follows:

$$\phi[\epsilon] = \phi \qquad \phi[(X = \nu X.\psi) \cdot U] = (\phi\{\nu X.\psi/X\})[U]$$

- a *store* C, used for accumulating structural constraints during symbolic execution.

We use $\varnothing_{H,m}$, $\varnothing_U$ and $\varnothing_C$ to denote the single-element history stack (m, ϵ) and the empty fixed-point stack and store, respectively.

For a given closed behavioural formula ϕ and method m, we construct a maximal tableau with root $\vdash_{\varnothing_{H,m},\varnothing_U,\varnothing_C} \phi$ that induces a set of structural formulae through its leaves, as described below. We denote the set of induced structural

formulae for ϕ and m with $\pi_m(\phi)$. We then define the translation of ϕ *w.r.t.* a given interface I:

$$\Pi_I(\phi) = \{ \textstyle\bigwedge_{m \in I^+} \chi_m \mid \chi_m \in \pi_m(\phi) \}$$

During tableau construction, the history stack, fixed-point stack and store are updated as follows, provided the current sequent is not a repeat of an earlier sequent (see below):

1. First, if ϕ is not a fixed-point formula, the propositional constant $\mathcal{S}(\phi)$ tagging the behavioural property ϕ of the current sequent is appended to the end of the frame of the top element of H;
2. Next,
   - if the behavioural property ϕ prescribes an internal transfer, then ε is appended to the end of the frame of the top element of H;
   - if ϕ prescribes a call from a to b, and the top element of H is in method a, then b is added at the end of the frame of the top element of H, and a new element (b, ϵ) is pushed onto H;
   - if ϕ prescribes a return from a to b, the top element of H is in method a and the next element is in method b, then a new structural constraint is added to the store, reflecting the possibility of currently not being at a return point, and the top element is popped from H; and
   - if ϕ is a fixed-point formula $\nu X.\phi$, then a new equation $X = \nu X.\phi$ is pushed onto the fixed-point stack U, if not already there; this conditional appending is denoted by $(X = \nu X.\phi) \circ U$.

Notice that non-emptiness of the history stack and closedness of $\phi[U]$ are invariants of the tableau construction.

Tableau System. The tableau system is given in Figure 2 as a set of goal-directed rules. Axioms are presented as rules with an empty set of premises, denoted by '$-$'. The condition $\mathsf{Ret}(i, a, b, H)$ used in the return rules is defined as $i = a \wedge H \neq \epsilon \wedge \exists F, H'.H = (b, F) \cdot H'$, *i.e.,* control is currently in method a, the call stack is not empty, and the control point on the top of the stack is in method b. Formally, a *tableau* $\mathcal{T} = (T, \lambda)$ is a tree T equipped with a labelling function λ mapping each tree node to a triple consisting of a sequent, a rule name (of the rule applied to this sequent), and a set of triples of shape (i, F, q) where q are literals (that is, atomic propositions in positive or negated form or propositional variables). The triple sets are non-empty only at applications of axiom rules; such leaves are termed *contributing*, and the set of triples is depicted (by convention) as a premise to the rule. A tableau for formula ϕ and method m is a tree with root $\vdash_{\varnothing_{H,m}, \varnothing_U, \varnothing_C} \phi$ obtained by applying the rules. A tableau is termed *maximal* if all its leaves are axioms.

If in a tableau there is a leaf node $\vdash_{(i,F)\cdot H, U, C} \phi$ for which there is an internal node $\vdash_{(i,F')\cdot H', U', C'} \phi$ such that F' is a prefix of F, U' is a suffix of U, and C' is a subset of C, we term the former node a *pseudo-repeat*; any node of the latter kind we term a *companion*. An internal tableau node is said to be *stable* if all its

$$p\ \frac{\vdash_{(i,F)\cdot H,U,C} p}{\{(i,F,p)\}\cup\{(i',F',\mathsf{ff})\mid(i',F')\in H\}\cup C} \qquad \neg p\ \frac{\vdash_{(i,F)\cdot H,U} \neg p}{\{(i,F,\neg p)\}\cup\{(i',F',\mathsf{ff})\mid(i',F')\in H\}\cup C}$$

$$\nu X\ \frac{\vdash_{(i,F)\cdot H,U,C} \nu X.\phi}{\vdash_{(i,F)\cdot H,(X=\nu X.\phi)\circ U,C} X} \qquad X\ \mathrm{unf}\ \frac{\vdash_{(i,F)\cdot H,U,C} X}{\vdash_{(i,F\cdot\mathcal{S}(X))\cdot H,U,C} \phi}\ (X=\nu X.\phi)\in U$$

$$\wedge\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi_1\wedge\phi_2}{\vdash_{(i,F\cdot\mathcal{S}(\phi_1\wedge\phi_2))\cdot H,U,C} \phi_1 \qquad \vdash_{(i,F\cdot\mathcal{S}(\phi_1\wedge\phi_2))\cdot H,U,C} \phi_2} \qquad \tau\ \frac{\vdash_{(i,F)\cdot H,U,C} [\tau]\phi}{\vdash_{(i,F\cdot\mathcal{S}([\tau]\phi)\cdot\varepsilon)\cdot H,U,C} \phi}$$

$$\mathrm{call}_0\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{call}\ b]\phi}{-}\ i\neq a \qquad \mathrm{call}_1\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{call}\ b]\phi}{\vdash_{(b,\epsilon)\cdot(i,F\cdot\mathcal{S}([a\ \mathsf{call}\ b]\phi)\cdot b)\cdot H,U,C} \phi}\ i=a$$

$$\mathrm{ret}_0\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{ret}\ b]\phi}{-}\ \neg\mathsf{Ret}(i,a,b,H) \qquad \mathrm{ret}_1\ \frac{\vdash_{(i,F)\cdot H,U,C} [a\ \mathsf{ret}\ b]\phi}{\vdash_{H,U,C\cup\{(i,F,\neg r)\}} \phi}\ \mathsf{Ret}(i,a,b,H)$$

$$\mathrm{IRep}\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi}{\{(i,F,\mathcal{S}(\phi))\}\cup C}\ \mathsf{IntRep}(\mathcal{S}(\phi),(i,F)\cdot H) \quad \mathrm{CRep}\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi}{-}\ \mathsf{CallRep}(\mathcal{S}(\phi),(i,F)\cdot H,c)$$

$$\mathrm{RRep}\ \frac{\vdash_{(i,F)\cdot H,U,C} \phi}{-}\ \mathsf{RetRep}(\mathcal{S}(\phi),(i,F)\cdot H,c)$$

Fig. 2. Tableau system

descendant leaves are axioms or pseudo-repeats. A tableau is stable if its root node is stable.

Tableau construction proceeds as follows. First, a minimal stable tableau is computed, *i.e.,* if a node is a pseudo-repeat, it is not further explored. If all pseudo-repeats in this tableau satisfy some repeat condition for any of their companions (see below), the tableau is maximal and construction is complete. Otherwise, all pseudo-repeats that are not satisfying any of the repeat conditions are simultaneously unfolded, using a breadth-first exploration strategy, and tableau construction continues until the tableau is stable again, upon which the checking for the repeat conditions is repeated. This process is guaranteed to terminate, as we state later, resulting in a finite maximal tableau.

Repeat Conditions. We now formulate the three repeat conditions used in the tableau system, giving rise to three types of repeat nodes. Only repeats of the first type, *i.e.,* internal repeats, contribute to triples, giving rise to recursion in structural formulae. In contrast, the other two repeat conditions only recognise that a similar situation has been reached before, and thus no new information will be obtained by further exploration. The first repeat condition requires merely the examination of the top frame of the history stack of the current sequent; the second requires the examination of the whole path from the root to the pseudo-repeat; while the third requires the examination of all remaining paths.

Internal repeat. Tableau construction guarantees that every tableau node of shape $\vdash_{H'\cdot(i,F'\cdot\mathcal{S}(\phi)\cdot F'')\cdot H'',U,C}\ \phi$ possesses an ancestor node $\vdash_{(i,F')\cdot H'',U',C'}\ \phi$ such that U' is a suffix of U and C' is a subset of C. As a consequence, every node of shape $\vdash_{(i,F'\cdot\mathcal{S}(\phi)\cdot F'')\cdot H,U,C}\ \phi$ is a pseudo-repeat (with some ancestor node of shape $\vdash_{(i,F')\cdot H,U',C'}\ \phi$ as companion); such pseudo-repeats are termed *internal repeat*s. Intuitively, an internal repeat indicates that a regularity in the

structure of method i has been discovered, and thus this regularity should be reflected in the structural formulae. Therefore, in this case $(i, F' \cdot \mathcal{S}(\phi) \cdot F'', \mathcal{S}(\phi))$ is added to the triple set of the IRep axiom. (Notice that in fact the propositional constant $\mathcal{S}(\phi)$ is mapped to a fresh propositional variable, here, and in the construction of the structural formulae. However, for clarity of presentation, we overload the symbols themselves, as their intended meaning should be clear from the context.)

Call repeat. A pseudo-repeat $\vdash_{(i,F)\cdot H,U,C} \phi$, which has an ancestor node as companion but is not an internal repeat, is a *call repeat* if H matches the call stack of the companion upto the latter's *return depth* (where matching means that the same methods are on the stack, with identical frames); in the special case where both stacks are shorter than the return depth, they have to be identical.

The return depth of a node is only defined if the subtableau of the companion is complete (*i.e.,* the pseudo-repeat is the only open branch). When we construct a tableau for a formula with multiple fixed-points, it can happen that two pseudo-repeats occur in the subtableaux of their respective companions. In this case, if both nodes are call repeats exploration terminates (for the current return depth); otherwise, by virtue of the tableau construction, the pseudo-repeat that is not a call repeat will never become one when continuing the tableau construction. Therefore, we can explore this node further, and break the mutual dependency.

The return depth of a tableau node n, denoted $\rho(n)$, is defined as the maximal difference between the number of applied return rules and the number of applied call rules on any path from n to a descendant node. Formally, where r and δ range over rule names and sequences of rule names, respectively, while $rules(\pi)$ denotes the sequence of rule names along a tableau path π:

$$\rho'(\epsilon) = 0 \qquad \rho'(r \cdot \delta) = \begin{cases} \rho'(\delta) + 1 & \text{if } r \in \{\mathsf{ret}_0, \mathsf{ret}_1\} \\ \rho'(\delta) - 1 & \text{if } r \in \{\mathsf{call}_0, \mathsf{call}_1\} \\ \rho'(\delta) & \text{otherwise} \end{cases}$$

$$\rho(n) = max\,\{\rho'(rules(\pi)) \mid \pi \text{ a path from } n \text{ to a descendant node}\} \cup \{0\}$$

Return repeat. A pseudo-repeat is called a *return repeat* if it has a companion on a different path from the root, such that its history stack is identical to the one of the companion.

Formally, the repeat conditions are defined as follows, where X is $\mathcal{S}(\phi)$, and c is the companion node of the pseudo-repeat with history stack H_c.

$$\begin{aligned} \mathsf{IntRep}(X, (i,F) \cdot H) &\Leftrightarrow X \in F \\ \mathsf{CallRep}(X, (i,F) \cdot H, c) &\Leftrightarrow X \notin F \wedge \mathsf{take}(\rho(c)+1, (i,F) \cdot H) = \mathsf{take}(\rho(c)+1, H_c) \\ \mathsf{RetRep}(X, (i,F) \cdot H, c) &\Leftrightarrow (i,F) \cdot H = H_c \end{aligned}$$

Termination. The repeat conditions ensure termination of tableau construction.

Theorem 1. *Maximal tableaux are finite.*

The proof of this and the remaining results can be found in [11].

3.2 Structural Formulae Induced by a Tableau

A maximal tableau for ϕ and m induces, through the sets of triples accumulated in the leaves, a set of structural formulae $\pi_m(\phi)$ in the following manner:

1. Let $\mathcal{L}$ be the set of non-empty triple sets collected from the leaves of the tableau. Build a collection of *choice sets* $\Lambda(\mathcal{L})$, by choosing one triple from each element in $\mathcal{L}$.
2. For each choice set $\lambda \in \Lambda(\mathcal{L})$,
   (a) Group the triples of λ according to method names: for each $i \in I$, define

$$\Xi_i = \{(F, q) \mid (i, F, q) \in \lambda\}$$

   (b) For each $i \in I$ such that $\Xi_i \neq \varnothing$, build a formula $i \Rightarrow \Omega(\Xi_i)$, where

$$\begin{array}{l} \Omega(\Xi) = \bigwedge_{\phi \in \Omega'(\Xi)} \phi \\ \Omega'(\Xi) = \{[a]\,\Omega(\Xi') \mid a \in I^- \wedge \Xi' = \{(F, q) \mid (a \cdot F, q) \in \Xi\} \wedge \Xi' \neq \varnothing\} \cup \\ \qquad \{\nu X.\Omega(\Xi') \mid X \in \mathcal{C}onst \wedge \Xi' = \{(F, q) \mid (X \cdot F, q) \in \Xi\} \wedge \Xi' \neq \varnothing\} \cup \\ \qquad \{q \mid (\epsilon, q) \in \Xi\} \end{array}$$

   (c) The *induced formula* χ for λ is the conjunction of the formulae obtained in the previous step.
3. The set $\pi_m(\phi)$ is the set of induced formulae for $\lambda \in \Lambda(\mathcal{L})$.

For example, the choice set $\lambda = \{(a, X \cdot b, \neg r), (a, X \cdot b, X)\}$ induces (by step 2) the structural formula $a \Rightarrow \nu X.\,[b]\,(\neg r \wedge X)$. Notice that all induced formulae are closed and guarded whenever the original behavioural one is.

Example 3. Consider formula $\phi = \nu X.\,[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$, *i.e.,* for every program execution consisting of consecutive sequences of calls from a to b followed by a return, the points at which control resumes in a are never return points themselves. Figure 3 shows the mapping $\mathcal{S}$ from the subformulae of ϕ to propositional constants, and the tableau that is constructed for this formula. The first node where a triple is produced is the one labelled ret_1; the triples then propagated to the two leaves that result from application of the rule for atomic propositions, and simple repeat, respectively. The tableau has two leaves with non-empty triple sets; $\mathcal{L}$ thus consists of two sets of two triples each.

Thus, to construct the set of structural formulae, we compute structural formulae for the four choice sets resulting from $\mathcal{L}$:

$$\begin{array}{l} \{(a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, \neg r), (a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, X_4)\} \\ \{(a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, \neg r), (b, X_4 \cdot X_1, \neg r)\} \\ \{(b, X_4 \cdot X_1, \neg r), (a, X_4 \cdot X_1 \cdot X_2 \cdot b \cdot X_5, X_4)\} \\ \{(b, X_4 \cdot X_1, \neg r)\} \end{array}$$

The first set gives rise to the structural formula $a \Rightarrow \nu X_4.\nu X_1.\nu X_2.\,[b]\,\nu X_5.(\neg r \wedge X_4)$, which simplifies to $\chi_1 = a \Rightarrow \nu X.[b](\neg r \wedge X)$: in the text of a, no initial sequence of consecutive call-to-b instructions ends in a return instruction. The

X_0	$\nu X.\,[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$	X_2	$[a\ \mathsf{call}\ b]\,X$	X_4	X	X_6	$\neg r$
X_1	$[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$	X_3	$[b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$	X_5	$\neg r \wedge X$		

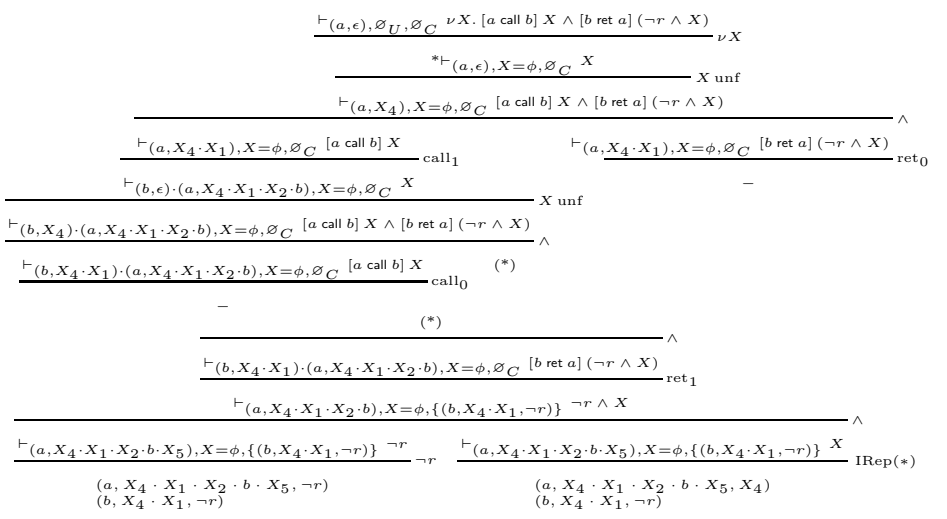


Fig. 3. Tableau for $\nu X.\,[a\ \mathsf{call}\ b]\,X \wedge [b\ \mathsf{ret}\ a]\,(\neg r \wedge X)$ and a, giving rise to $\{a \Rightarrow \nu X.\,[b]\,(\neg r \wedge X), b \Rightarrow \neg r\}$

last set gives rise to the formula $\chi_2 = b \Rightarrow \neg r$ (again after simplification): the text of b does not begin with a return instruction. The formulae constructed from the second and third set are subsumed by χ_2, and hence $\pi_a(\phi) = \{\chi_1, \chi_2\}$. For ϕ and method b there is a single tableau, which has no leaf triples, and hence $\pi_b(\phi) = \{\mathsf{tt}\}$. Thus, $\Pi(\phi) = \{\chi_1, \chi_2\}$.

More property translations and example tableaux illustrating the various repeat conditions can be found in [11].

3.3 Correctness of the Translation

Because of space limitations, we cannot present here the full soundness and completeness proofs; instead, we refer to the accompanying report [11]. The main idea is the construction of a *proof system* that allows to show that a structural formula χ implies a behavioural formula ϕ. The rules of the proof system stand in a one-to-one correspondence with the rules of the tableau system, except for the handling of fixed-point formulae, which are unfolded (so proof trees can be infinite). The proof tree that corresponds to the unfolding of a tableau for behavioural formula ϕ inducing structural formula χ, constitutes a legal proof of χ implying ϕ (*i.e.,* soundness of Π). Moreover, any formula χ' that implies ϕ is subsumed by a formula induced by the tableau (*i.e.,* completeness of Π).

Theorem 2. *Translation Π from behavioural to structural formulae is sound and complete.*

4 Application: Compositional Verification

The original motivation for the present work has been the wish to extend an earlier developed compositional verification method [12] to behavioural properties. The compositional verification method is based on the computation of maximal models: a model is said to be *maximal* for a given property ϕ, if it satisfies ϕ and simulates (*w.r.t.* a property-preserving simulation relation) all other models satisfying ϕ. Due to the close connection between simulation and satisfaction in our logic, we obtain the following compositional verification principle: showing $\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi$ can be reduced to showing $\mathcal{G}_1 \models \phi$ (*i.e.*, component $\mathcal{G}_1$ satisfies a *local assumption* ϕ) as long as $\mathcal{G}_\phi \uplus \mathcal{G}_2 \models \psi$ (*i.e.*, component $\mathcal{G}_2$, when composed with the maximal flow graph $\mathcal{G}_\phi$ for ϕ, satisfies the *global guarantee* ψ).

Thus, the compositional verification problem is reduced to finding maximal flow graphs. However, given a property ϕ over a flow graph (behaviour), there is no guarantee that the maximal model of ϕ is a valid flow graph (behaviour). At the structural level this problem can be solved, because we can precisely characterise legal flow graphs *w.r.t.* an interface I by a structural formula θ_I in our logic. Then, if ϕ is an arbitrary structural formula, the maximal model of the formula $\phi \wedge \theta_I$ is a flow graph $\mathcal{G}_{\phi,I}$ which represents all flow graphs with interface I that satisfy ϕ.

However, there is no such way to precisely characterise flow graph behaviour in our logic (*cf.* [12]), and thus one cannot directly apply the above compositional verification principle to behavioural properties. In [12], we proposed a "mixed" rule where global guarantees are behavioural, but local assumptions are structural. With the results of the present paper, however, this rule can be combined with the characterisation (1) to yield the following sound and complete compositional verification principle, where both the global guarantee (required to be disjunction-free) and the local assumption are behavioural.

$$\frac{\mathcal{G}_1 \models_b \phi \qquad \left\{\mathcal{G}_{\chi, I_{\mathcal{G}_1}} \uplus \mathcal{G}_2 \models_b \psi\right\}_{\chi \in \Pi_{I_{\mathcal{G}_1}}(\phi)}}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_b \psi} \quad \mathcal{G}_1 \text{ closed}$$

Notice that when applying the rule, instead of showing $\mathcal{G}_1 \models_b \phi$ it suffices to show $\mathcal{G}_1 \models_s \chi$ for some $\chi \in \Pi_{I_{\mathcal{G}_1}}(\phi)$. Completeness of the principle guarantees that no *false negatives* are possible: if the second premise fails, then there is indeed a legal flow graph $\mathcal{G}$ with interface $I_{\mathcal{G}_1}$ such that $\mathcal{G} \models_b \phi$ but $\mathcal{G} \uplus \mathcal{G}_2 \not\models_b \psi$.

An alternative way of using characterisation (1) for compositional verification is to apply it to the global guarantee (see [11]).

5 Conclusions and Future Work

This paper presents a precise characterisation of (disjunction-free) behavioural formulae as sets of structural formulae, in a context where programs are abstracted as flow graphs, and properties are expressed in a fragment of the modal μ-calculus with boxes and greatest fixed points only. As one significant application, we state a sound and complete *compositional verification* principle for

behavioural properties based on maximal models. Another possible application is the reduction of infinite-state verification of behavioural control flow properties to finite-state verification of structural properties.

Extensions. Unlike the other connectives of the logic, validity of sequents is not compositional *w.r.t. disjunction* in our tableau system. Disjunction can still be handled, though at the expense of completeness, by adding two symmetric tableau rules that simply "drop" the right respectively the left disjunct. A behavioural formula and a method will thus give rise to a set of tableaux, for which we take the union of their induced sets of structural formulae. Alternatively, to potentially obtain a complete translation (if such exists), we plan to generalise the sequent format, *e.g.,* in the style of Gentzen sequents, and then also tableau construction and formula extraction. We also plan to study whether the characterisation can be extended for the logic with diamonds and least fixed points, and for richer program models (*e.g.,* with exceptions, or multithreading, as in [13]), and whether the compositional verification principle can be generalised to *open* components. For the last extension, two different approaches will be considered: (i) the translation is generalised to formulae over open interfaces, requiring the generalisation of Definition 6 for open flow graphs, and (ii) every open component is "closed" by composing it with a *most general environment* before the characterisation is applied.

Implementation. An implementation of the translation has been developed in Ocaml, and is available via a web-based interface [10]. It returns a tableau per method, plus a set of structural formulae (after applying some basic logical simplifications, *e.g.*, removing unused fixed-points, to make the output more readable). It has been applied on all examples in the paper and the accompanying technical report [11]. In all cases, the output is produced within seconds. Various *optimisations* of the translation are possible. For instance, since logically subsumed formulae are redundant in the characterisation, the construction of choice sets can be optimised as follows: if a triple is picked from a contributing leaf, then the same triple must be selected from all other contributing leaves containing it.

In future work, the *complexity* of the tableau construction will be studied by finding upper bounds for the size of generated tableaux, and for the number and size of generated formulae.

References

1. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: Logic in Computer Science (LICS 2007), Washington, DC, USA, pp. 151–160. IEEE Computer Society Press, Los Alamitos (2007)
2. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM TOPLAS 27, 786–818 (2005)
3. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic for nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

4. Bouajjani, A., Fernandez, J.C., Graf, S., Rodriguez, C., Sifakis, J.: Safety for branching time semantics. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 76–92. Springer, Heidelberg (1991)
5. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 545–623. North-Holland, Amsterdam (2000)
6. Dam, M., Gurov, D.: μ-calculus with explicit points and approximations. Journal of Logic and Computation 12(2), 43–57 (2002)
7. Dams, D., Namjoshi, K.S.: The existence of finite abstractions for branching time model checking. In: Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2004), pp. 335–344. IEEE Computer Society Press, Los Alamitos (2004)
8. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
9. Grumberg, O., Long, D.: Model checking and modular verification. ACM TOPLAS 16(3), 843–871 (1994)
10. Gurov, D., Huisman, M.: From behavioural to structural properties: A tool web interface, `http://www.csc.kth.se/~dilian/Projects/CVPP/beh2struct.php`
11. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. Technical Report TRITA-CSC-TCS 2007:3, KTH Royal Institute of Technology, Stockholm, 35 pages (2007), `http://www.csc.kth.se/~dilian/Papers/techrep-07-3.pdf`
12. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
13. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: International Conference on Formal Engineering Methods (ICFEM 2008). LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
14. Huisman, M., Gurov, D.: Composing modal properties of programs with procedures. In: Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2007). Electronic Notes in Theoretical Computer Science (to appear, 2008)
15. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
16. Reddy, U., Kamin, S.: On the power of abstract interpretation. Computer Languages 19(2), 79–89 (1993)
17. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58(1-2), 206–263 (2005)
18. Schneider, F.B.: Enforceable security policies. ACM Trans. Infinite Systems Security 3(1), 30–50 (2000)
19. Schöpp, U., Simpson, A.K.: Verifying temporal properties using explicit approximants: Completeness for context-free processes. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 372–386. Springer, Heidelberg (2002)

Query-Driven Program Testing*

Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith

Formal Methods in Systems Engineering, FB Informatik, TU Darmstadt
{holzer,schallhart,tautschnig,veith}@forsyte.de

Abstract. We present a new approach to program testing which enables the programmer to specify test suites in terms of a versatile query language. Our query language subsumes standard coverage criteria ranging from simple basic block coverage all the way to predicate complete coverage and multiple condition coverage, but also facilitates on-the-fly requests for test suites specific to the code structure, to external requirements, or to ad hoc needs arising in program understanding/exploration. The query language is supported by a model checking backend which employs the CBMC framework. Our main algorithmic contribution is a method called *iterative constraint strengthening* which enables us to solve a query for an arbitrary coverage criterion by a single call to the model checker and a novel form of incremental SAT solving: Whenever the SAT solver finds a solution, our algorithm compares this solution against the coverage criterion, and strengthens the clause database with additional clauses which exclude redundant new solutions. We demonstrate the scalability of our approach and its ability to compute compact test suites with experiments involving device drivers, automotive controllers, and open source projects.

1 Introduction

In industrial software engineering, program testing is to remain the pivotal debugging and validation technology. While randomized and directed testing are important to achieve global assurance about software quality, and model-based testing helps to verify the conformance of the program with a high-level specification, there is practical need for a source code oriented white box testing methodology which assists the programmer in the software engineering cycle. Such a methodology should provide seamless support for code-driven testing, i.e., exploration of code under development, and requirement-driven testing for systematic quality assertion.

To address this need, we introduce a query language which combines easy navigation in real life C code with the ability to formulate complex coverage criteria. Providing straightforward queries for standard coverage criteria, our language FQL (FSHELL query language) aims to strike the right balance between expressiveness and simplicity. In FQL, a *program query* asks for a test suite following the general form

> **in** *prefix* **cover** *goals* **passing** *scope*

where the optional *prefix* directs the query to a specific program part, e.g., a source file or a function, *goals* describes the coverage criterion to be fulfilled by the test suite, and

* Supported by DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1).

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 151–166, 2009.

Listing 1. C source code of example `bla.c` with program counters

```
1 int cmp(int x, int y) {
2     int [1]value = 0;
3     if [2](x > y) [3]value = 1;
4     else if [4](x < y) [5]value = −1;
5     [6]return value;
6 }

8 int main(int argc, char* argv []) {
9     int x, y, z, xy, yz, xz;
10    [8]x = [7]input();
11    [10]y = [9]input();
12    [12]z = [11]input();
13    [14]xy = [13]cmp(x, y);
14    [16]yz = [15]cmp(y, z);
15    [18]xz = [17]cmp(x, z);
16    if [22]([19]xy == 1 && [20]yz == 1
17        && [21]xz != 1)
18 ERROR: [23]err();
19    [24]return 0;
20 }
```

an optional *scope* restricts test cases to pass through certain program paths only. For example, the query

```
> in /bla.c/cmp/ cover @blocks passing @call(err)
```

calls for a test suite which covers all blocks in function `cmp` of file `bla.c`, such that in all test cases, a call to `err` inside `cmp` is performed[1]. Other important and classical coverage criteria such as *predicate coverage*, *condition coverage*, *decision coverage*, *modified condition/decision coverage*, *predicate complete coverage* [1] etc. can also be expressed by natural queries in our language.

The added value of our language lies in the ability to define the query scope and the query goals in a quite flexible manner. Suppose for instance that in Listing 1 we want to cover (1) all calls to `cmp` and (2) all decisions inside `cmp`. This amounts to a coverage criterion which combines basic block coverage for (1) and decision coverage for (2). There are two different possibilities for this combination: either we want to cover the union of all call positions and all decisions, and write the query

```
> in /bla.c/ cover @call(cmp), cmp/@decisions
```

or we want to cover all possible *combinations* of calls to `cmp` and subsequent decisions inside `cmp`, i.e., the Cartesian product of the individual test goals:

```
> in /bla.c/ cover @call(cmp) -> cmp/@decisions
```

To ensure that, for each call site, each of the decisions is reached before the body of `cmp` is left, we write

```
> in /bla.c/ cover @call(cmp) -[@func(cmp)\@exit]> cmp/@decisions
```

Solutions to these queries are test suites. In case of Listing 1, these can be most easily described as sets of triples with input values for the variables x, y, z. For the first query, the singleton test suite $\{(1,0,1)\}$ covers all blocks and decisions in Listing 1; for the second and third case, possible solutions are $\{(1,0,1),(2,0,1),(1,0,2)\}$ and $\{(1,0,0),(0,1,0),(0,0,1)\}$ respectively. Note that the first test suite does not satisfy the second and third queries and the second suite does not satisfy the last query; the third solution, however, satisfies all three queries.

[1] Expressions starting with "@", such as `@blocks`, typically denote sets of program locations, see Section 3.1 for a detailed explanation.

In Section 2 we build the mathematical foundation to formulate testing requirements as queries. In particular, we show how to state a query as a pair $\langle A, Q \rangle$ of automata over predicates. In this pair, the *observation automaton A* corresponds to the *scope* to be explored, and the *test goal automaton Q* specifies the *goals* to be covered. In Section 3, we provide an overview of our query language FQL and show how to translate FQL queries into such a pair $\langle A, Q \rangle$. Thus, the language reduces to a simple mathematical core in which we are able to formulate all relevant coverage criteria in a uniform way.

The second major contribution of this paper is an efficient query engine which integrates our theoretical framework, code instrumentation, bounded model checking, and SAT enumeration into a tool of high efficiency. Our query engine employs and adopts the software model checking framework of Kroening's CBMC [2]. Given a query $\langle A, Q \rangle$, our tool performs the following conceptual steps, cf. Figure 1:

(1) We instrument the source code with monitors derived from the observation automaton A and the test goal automaton Q in such a way that the states of the monitors reflects the automata states.

(2) We use the code base of CBMC to obtain a SAT instance ϕ whose solutions correspond to the program paths π in the scope given by A. The instrumentation of step (1) guarantees that for each solution, we can easily determine which goals of Q are covered.

(3) We use the SAT solver to enumerate test cases as solutions to the SAT instance until we satisfy the coverage criterion defined by the query. The *iterative constraint strengthening* technique (ICS) used in this step is discussed below.

(4) To remove redundant test cases, we perform a test suite minimization. In our current implementation, we only do a simple post-processing; in future work, we plan to implement more aggressive minimization strategies. Note that the ICS enumeration in (3) involves nondeterministic choices which may give leverage to accelerate the algorithm with suitable heuristics.

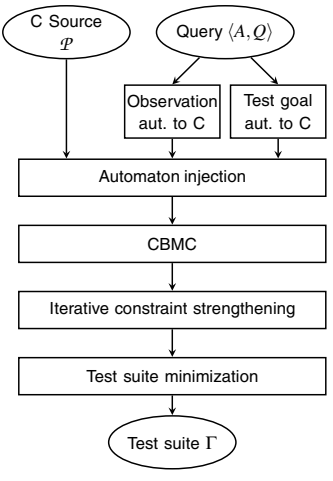


Fig. 1. Query processing

Iterative Constraint Strengthening. A naive implementation of step (3) above would either use SAT enumeration to compute an enormous number of test cases until the test goals are reached, or it would call the SAT solver for each query goal anew. In *iterative constraint strengthening (ICS)*, we circumvent both problems by modifying the clause database of the SAT solver on-the-fly. Whenever the SAT solver halts to output a solution, we compare the test case obtained from this solution against the test goals. Then we add new clauses to the clause database in such a way that the next solution is guaranteed to satisfy at least one hitherto uncovered test goal. In this way, we exploit incremental SAT solving to quickly enumerate a test suite of high quality: Since we only add new clauses to the clause database, the SAT solver is able to reuse information learned in prior invocations. A similar strategy is used in *groupwise constraint strengthening (GCS)*, a further refinement of ICS. In GCS, we address coverage criteria such

as multiple condition coverage or predicate complete coverage which have a nominally exponential number of test goals by partitioning these goals into a small number of groups characterized by a common compound goal.

We show that FSHELL has better practical performance than BLAST's test case generation facility [3]: On comparable hardware, our test suites are computed faster, and contain fewer test cases. Due to the minimization step, our results also improve on those reported in our previous tool paper [4].

Note that our choice of CBMC and bounded model checking as a query solving backend has advantages which come at a price: On the one hand, we achieve excellent performance and have the guarantee that the model-checker respects ANSI-C, which is important for low level code, our primary application area. On the other hand, a bounded model checking approach may be *unable to compute certain test cases* involving paths larger than the constant bound. It is easy to come up with examples where this situation will happen, but it is is detectable by CBMC and accounted for in our implementation; it has neither occurred in the experiments we did for comparison with BLAST, nor in our experiments based on real-life controller code. In future work, we plan to complement the CBMC backend with abstraction-based and randomized test case generation backends.

Related Work. Beyer et al. [3] use the C model checker BLAST [5] for test case generation, focusing on basic block coverage only. BLAST has a two level specification language [6]. On a low level they specify trace properties by observer automata written in a C-like manner. On a high level they relate these automata by reachability queries. In contrast to FSHELL, their language is tailored towards verification. Furthermore, BLAST is based on predicate abstraction whereas CBMC is a SAT-based bounded model checker. As our experiments show, we outperform BLAST regarding test case generation. Lee et al. [7,8] investigate test case generation with model checkers giving coverage criteria in temporal logics. Java PathFinder [9] and SAL2 [10] use model checkers for test case generation, but they do not support C semantics.

2 A Formal Testing Framework

Given a program $\mathcal{P}$, we consider the possibly infinite *transition system* $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$ induced by $\mathcal{P}$ which consists of the state space $\mathcal{S}$, a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and a non-empty set of initial states $I \subseteq \mathcal{S}$.

Definition 1 (State Sequences and Paths). *Given a transition system $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$, a* state sequence *is a finite and non-empty word $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^+$ of states $s_i \in \mathcal{S}$. The sequence π is a* path*, if $\langle s_i, s_{i+1} \rangle \in \mathcal{R}$ holds for all $1 \leq i < n$ and if $s_1 \in I$. For a state $s \in \mathcal{S}$, we write $s \in \pi$, iff $s = s_i$ holds for some $1 \leq i \leq n$, and we denote with $\Pi^{\mathcal{T}} \subseteq \mathcal{S}^+$ the* set of paths *of $\mathcal{T}$.*

We use *state predicates* to describe properties of individual program states and we use *path* and *path set predicates* in the description of individual test goals and coverage criteria.

Definition 2 (State, Path, & Path Set Predicates). *Given a transition system $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$, we define a* state predicate *p as a predicate on the state space $\mathcal{S}$, a* path

predicate ϕ *as a predicate over the set* $\Pi^{\mathcal{T}}$*, and a* path set predicate Φ *as a predicate over the sets of paths* $2^{\Pi^{\mathcal{T}}}$*. We write* $s \models p$ *iff a state* $s \in \mathcal{S}$ *satisfies* p*,* $\pi \models \phi$ *iff a path* $\pi \in \Pi^{\mathcal{T}}$ *satisfies* ϕ*, and* $\Gamma \models \Phi$ *iff a path set* $\Gamma \subseteq \Pi^{\mathcal{T}}$ *satisfies* Φ*.*

We call a state predicate p, a path predicate ϕ, or a path set predicate Φ *feasible over* $\mathcal{T}$*,* iff, respectively, there exists a state $s \in \mathcal{S}$ with $s \models p$, there exists a path $\pi \in \Pi^{\mathcal{T}}$ with $\pi \models \phi$, and there exists a path set $\Gamma \subseteq \Pi^{\mathcal{T}}$ with $\Gamma \models \Phi$. Frequently, we are looking for a path (path set) which *contains* a state (a path) which satisfies a given state (path) predicate—leading to an *implicit existential quantification:*

Definition 3 (Implicit Existential Quantification). *To evaluate a state predicate* p *over a path* π*, we implicitly interpret* p *to be existentially quantified, i.e.,* $\pi \models p$ *stands for* $\exists s \in \pi . s \models p$*. Analogously, a path predicate* ϕ *is existentially evaluated over a path set* Γ*, i.e.,* $\Gamma \models \phi$ *iff* $\exists \pi \in \Gamma . \pi \models \phi$*.*

Remark 1. Note that a path π can satisfy a state predicate p *and* its negation $\neg p$, if there exist two states $s, s' \in \pi$ with $s \models p$ and $s' \models \neg p$. Moreover, a state predicate p can also be interpreted over a path set Γ in the natural way, i.e., $\Gamma \models p$ iff $\exists \pi \in \Gamma . \exists s \in \pi . s \models p$.

Program Observations. We use sequences of state predicates (*traces*) to specify program paths. A trace matches a state sequence if each state in the sequence satisfies the corresponding predicate. A trace automaton is an automaton accepting traces; each trace in turn specifies a set of program paths.

Definition 4 (Traces and Trace Automata). *Let* P *be a finite set of state predicates and* $\mathcal{S}$ *be a state space. Then a* trace *is a finite non-empty word* $t = \langle t_1, \ldots, t_n \rangle \in P^+$*. A trace* matches *a state sequence* $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^+$ *(denoted with* $\pi \models t$*), iff* $s_i \models t_i$ *for all* $1 \leq i \leq n$*.*

A trace automaton *over* P *is a nondeterministic finite state automaton* A *accepting traces over the alphabet* P*. We write* $\mathcal{L}(A)$ *to denote the set of traces accepted by* A *and* $\mathsf{acc}(A)$ *to denote the set of accepting states of* A*. A trace automaton* A *over* P matches *a state sequence* π *(denoted with* $\pi \models A$*), iff there exists a trace* $t \in \mathcal{L}(A)$ *with* $\pi \models t$*.*

Remark 2. Although we have – for the sake of simplicity – defined trace automata as finite state automata, our framework naturally extends to other types of automata such as push-down automata for which we can construct C monitors, cf. Section 4.1.

We will use traces and trace automata as a natural tool for defining path predicates in the language FQL. In particular, we will employ trace automata for two distinct ends: First, as *observation automata* which *restrict the paths in* $\mathcal{T}$ to those required in a query; and second, as *test goal automata* which specify the individual test goals of a coverage criterion.

Definition 5 (Path Restriction by Observation Automata). *Let* $\mathcal{T}$ *be a transition system and* A *a trace automaton. Then we define the set of paths in* $\mathcal{T}$ *restricted by observation automaton* A *as* $\Pi_A^{\mathcal{T}} = \{\pi \in \Pi^{\mathcal{T}} \mid \pi \models A\}$*.*

Coverage Criteria. In the framework of this paper, we define a *test case* to be a single path in $\Pi_A^{\mathcal{T}}$ and a *test suite* as a subset of $\Pi_A^{\mathcal{T}}$. Correspondingly, a *coverage criterion* imposes a predicate on test suites:

Definition 6 (Test Case & Test Suite). *Let $\mathcal{T}$ be a transition system and let A be an observation automaton for $\mathcal{T}$. Then a* test case *for the set of paths $\Pi_A^{\mathcal{T}}$ is a single path $\pi \in \Pi_A^{\mathcal{T}}$ and a* test suite *Γ is a finite subset $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ of the paths in $\Pi_A^{\mathcal{T}}$.*

Definition 7 (Coverage Criterion). *A* coverage criterion *Φ is a mapping from a transition system $\mathcal{T}$ and an observation automaton A to a path set predicate $\Phi_A^{\mathcal{T}}$ over $2^{\Pi^{\mathcal{T}}}$. We say that $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ satisfies the coverage criterion Φ on $\mathcal{T}$ under the restriction A iff $\Gamma \models \Phi_A^{\mathcal{T}}$ holds.*

While our definition of coverage criteria is very general, most coverage criteria used in practice are based on lists of test goals which need to be satisfied. The test goals themselves are typically either state or path predicates. This prototypical setting is accounted for in the next definition.

Definition 8 ((State) Regular Coverage Criterion and Test Goals). *A* regular coverage criterion *Φ is a coverage criterion constructed in the following way:*

(i) There is a mapping $\Phi(\mathcal{T},A)$ which maps $\mathcal{T}$ and A to a list of test goals *$\Phi(\mathcal{T},A) = \{\Psi_1,\ldots,\Psi_k\}$.*
(ii) This mapping induces the coverage criterion $\Phi_A^{\mathcal{T}}$ as follows:

$$\Gamma \models \Phi_A^{\mathcal{T}} \text{ iff } \bigwedge_{i=1}^{k} \Pi_A^{\mathcal{T}} \models \Psi_i \Rightarrow \Gamma \models \Psi_i$$

Intuitively, this amounts to the following coverage criterion: "For each test goal which is feasible in $\Pi_A^{\mathcal{T}}$, the test suite Γ must contain a concrete test case."

Φ is a state regular coverage criterion*, if $\Phi(\mathcal{T},A)$ contains only state predicates.*

As an example, consider basic block coverage $\mathsf{BB}_A^{\mathcal{T}}$, which is a state regular coverage criterion: induced by the test goals $\mathsf{BB}(\mathcal{T},A) = \{\mathsf{block}_1,\ldots,\mathsf{block}_k\}$. Here k denotes the number of basic blocks in $\mathcal{T}$, and each predicate block_i holds true at the first statement of the i-th basic block in the program.

We will now define test goal automata which are used to specify the test goals needed in regular coverage criteria.

Definition 9 (Test Goal Automaton). *A* test goal automaton *Q is a trace automaton where each accepting state a gives rise to a test goal Ψ_a:*

$$\pi \models \Psi_a \text{ iff } \exists t.\, \pi \models t \text{ and } Q \text{ accepts } t \text{ in state } a$$

Thus, the test goal Ψ_a requires a path matched by a trace which Q accepts in state a. The test goal automaton Q naturally induces a regular coverage criterion $\mathsf{cov}[Q]$ based on the set $\mathsf{cov}[Q](\mathcal{T},A) = \{\Psi_a \mid a \in \mathsf{acc}(Q)\}$ of test goals.

Note that a single path may match more than one test goal simultaneously: First, each path is matched by a number of different traces, and second, more than one accepting state may be reached through a trace in Q.

We conclude this section with a formal definition of program queries, as introduced in Section 1.

Definition 10 (Program Query & Result). *A* program query $\langle A, Q \rangle$ *consists of an observation automaton A and a test goal automaton Q. A valid* result *to the query $\langle A, Q \rangle$ on transition system $\mathcal{T}$ is a test suite $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ with $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$.*

3 Syntax and Semantics of FQL

The FSHELL query language FQL facilitates the specification of test suites over C source code. To decouple the language from the algorithmic details of the query engine, and to provide leeway for different query solving backends, we designed FSHELL as a declarative language. FQL contains three layers which reflect the formal model of Section 2:

(i) state predicates over program variables and the program counter,
(ii) trace automata to express both observation automata and test goal automata, and
(iii) program queries to express coverage criteria.

In the following subsections, we will describe these layers along with examples referring to Listing 1. Due to length restrictions, the presentation of FQL is kept informal; we refer the reader to [11] for more details. Section 4 describes our query solving engine based on bounded model checking.

3.1 State Predicates

We have seen in Section 2 that sets of state predicates are at the center of our formal model. For instance, basic block coverage is induced by the set of test goals $\mathsf{BB}(\mathcal{T}, A) = \{\mathsf{block}_1, \ldots, \mathsf{block}_k\}$. FQL is therefore equipped with *predicate generators* to extract sets of predicates from the C source code, and to create new sets of predicates. For example, the predicate generator `@blocks` yields the set $\{\mathsf{block}_1, \ldots, \mathsf{block}_k\}$ of predicates. Note that each block_i has the form $\mathsf{pc} = \text{const}$ where pc is the program counter. Syntactically, all predicate generators are prefixed with "`@`". Semantically, a predicate generator either yields a set of predicates over the program counter pc, or a set of predicates over the program variables.

Many predicate generators are used to extract sets of predicates from the source code. Examples of such predicate generators include `@file(bla.c)` which captures all program counter values of statements in the source file `bla.c`, `@func(main)` which captures the statements in function `main`, `@line(3)` to capture the statements in line 3, `@call(cmp)` to match all function calls of `cmp`, and `@entry` as well as `@exit` which capture all function entry and exit points respectively. In case of Listing 1 we get, e.g., `@call(cmp)` $= \{\mathsf{pc} = 13, \mathsf{pc} = 15, \mathsf{pc} = 17\}$.

To introduce new predicates not present in the source code, we use the predicate generator `@new-pred(cond)`, where cond is an arbitrary side-effect free C expression. For example, `@new-pred(x <= 7)` generates a singleton set $\{x \leq 7\}$ of state predicates.

For certain coverage criteria such as MC/DC, we also need the predicate generator `@grouped-conditions` which generates a *set of sets*, where each inner set captures the program counter values of the individual predicates which constitute a decision. Returning to Listing 1, we have `@grouped-conditions` $= \{\{\mathsf{pc}=2\},\{\mathsf{pc}=4\},\{\mathsf{pc}=19,\mathsf{pc}=20,\mathsf{pc}=21\}\}$. To support the succinct formulation of most relevant coverage criteria, FQL contains a rich variety of predicate generators and can be easily extended with further ones without conceptual changes to the language [11].

Operations on Sets of State Predicates. Given two sets A and B of state predicates, FQL provides the following set-theoretic operations:

(and)	$A\&B \equiv \{a \wedge b \mid a \in A, b \in B\}$	$A,B \equiv A \cup B$	(union)
(or)	$A\|B \equiv \{a \vee b \mid a \in A, b \in B\}$	$A\backslash B \equiv A \setminus B$	(difference)
(negation)	$!A \equiv \{\neg a \mid a \in A\}$	$2\hat{}A \equiv \{A' \mid A' \subseteq A\}$	(powerset)

We add `big-and(`A`)` $\equiv \bigwedge_{a\in A} a$ and `big-or(`A`)` $\equiv \bigvee_{a\in A} a$ to describe the conjunction and disjunction of all elements of a set of state predicates. To apply an operation to each element in a set, or to *each set in a set of sets*, we introduce the `set()` operator. Moreover, `union()` forms a single set from a set of sets. Given a set S of sets and an operation $o(s)$ on a set s of state predicates, we define:

$$\texttt{set(}o(s)\ \texttt{:}\ s\ \texttt{in}\ S\texttt{)} \equiv \{o(s) \mid s \in S\} \qquad \texttt{union(}S\texttt{)} \equiv \bigcup_{s\in S} s$$

Operations on Conditions. In describing coverage criteria, *conditions* occurring in the source code play a crucial role. A condition is an atomic expression which is possibly combined with other conditions using &&, ||, and ! to compute the decision involved in executing an **if**, **for**, **while**, **switch** or ?: statement. The generator `@pred-wo-loc()` extracts conditions from source code locations identified by program counter values. In addition, `@predicate()` and `@neg-predicate()` conjoin the extracted conditions with the corresponding predicate over the program counter. For example, let $C = \{\mathsf{pc} = 19, \mathsf{pc} = 20, \mathsf{pc} = 21\}$ be such a set, referring to Listing 1. Then we have

$$\begin{aligned}
\texttt{@pred-wo-loc(}C\texttt{)} &= \{xy = 1, yz = 1, xz \neq 1\} \\
\texttt{@predicate(}C\texttt{)} &= \{\mathsf{pc} = 22 \wedge xy = 1, \mathsf{pc} = 22 \wedge yz = 1, \mathsf{pc} = 22 \wedge xz \neq 1\} \\
\texttt{@neg-predicate(}C\texttt{)} &= \{\mathsf{pc} = 22 \wedge xy \neq 1, \mathsf{pc} = 22 \wedge yz \neq 1, \mathsf{pc} = 22 \wedge xz = 1\}
\end{aligned}$$

Note that $\mathsf{pc} = 22$ refers to the location of the decision inside which the conditions in C occur.

State Regular Coverage Criteria. Besides simple test goals such as `@blocks`, FQL can can also describe more complex coverage criteria. We illustrate this feature on the example of *multiple condition coverage*. Recall that multiple condition coverage requires a test suite to cover—for each decision—all Boolean combinations of all conditions occurring in the respective decision. The test goals are therefore given by

```
union( set(
 union( set( big-and(@predicate(I) & @neg-predicate(D\I)) : I in 2^D) ):
D in @grouped-conditions ) )
```

Hierarchical Navigation. In practical queries, the predicate generators `@file(bla.c)`, `@line(3)`, `@func(foo)`, as well as `@entry` and `@exit` occur quite frequently. We therefore allow the following abbreviations which facilitate hierarchical navigation in the source code:

$$\texttt{/bla.c/} = \texttt{@file(bla.c)} \qquad \texttt{/bla.c/42} = \texttt{@file(bla.c)}\ \&\ \texttt{@line(42)}$$
$$\texttt{foo/} = \texttt{@func(foo)} \qquad \texttt{/bla.c/foo/} = \texttt{@file(bla.c)}\ \&\ \texttt{@func(foo)}$$
$$\texttt{foo/\^{}} = \texttt{@entry(foo)} \qquad \texttt{foo/\$} = \texttt{@exit(foo)}$$
$$\texttt{foo/}SP = \texttt{@func(foo)}\ \&\ SP \qquad \texttt{/bla.c/}SP = \texttt{@file(bla.c)}\ \&\ SP$$

In the last line, *SP* is to be replaced by any state predicate expression. Note that FQL also supports macros for frequently used expressions such as complex coverage criteria. Due to space restrictions we do not describe the macro feature in detail.

3.2 Trace Automata

Recall that trace automata are used to define path predicates, and to act as both observation automata and test goal automata. By implicit existential quantification, every state predicate can also be viewed as a path predicate, and it is easy to construct the corresponding automaton. Moreover, a set of state predicates naturally gives rise to an automaton with one accepting state for each state predicate in the set. For example, `@blocks` corresponds to an automaton with $|\texttt{@blocks}|$ accepting states, one for each basic block. The following list exemplifies the most important automata theoretic operations of FQL which enable the user to manipulate and combine trace automata explicitly: Let A_1, A_2, A_3 be trace automata:

$$A_1, A_2 \equiv A_1 \cup A_2 \qquad \text{(union)}$$
$$A_1\texttt{->}A_2 \equiv A_1 \circ \mathsf{true}^* \circ A_2 \qquad \text{(sequencing)}$$
$$A_1\texttt{-[}A_3\texttt{]>}A_2 \equiv A_1 \circ A_3^* \circ A_2 \qquad \text{(restricted sequencing)}$$

Consider for example `main/^->main/$` over Listing 1: the traces of this automaton will match those program executions which pass the exit of `main` (line 19). In contrast, `main/^-[@file(bla.c)\@label(ERROR)]>main/$` requires that between the entry and the exit of `main` only locations other than those labeled "ERROR" (line 18) are seen. Note that each of these operations corresponds to a specific automata theoretic construction. Due to the special role of accepting states in defining test goals, we cannot use the standard automata theoretic minimization techniques, cf. [11].

3.3 Program Queries

We are now ready to define the program queries introduced in Section 1. Let A and B be FQL expressions which can be interpreted as trace automata (i.e., either trace automata, or sets of predicates as explained in the previous section). Then **`cover` *Q* `passing`** *A* expresses the program query $\langle A, Q\rangle$ with the semantics given in Definition 10.

Recall from Section 1, that FQL queries can also have a prefix. This prefix restricts all state predicates to a certain program part, e.g., a certain file. It is easy to see that the prefix can be moved into A and Q. For example, a query such as

```
> in /bla.c/ cover @line(4),@call(cmp)
  passing @file(bla.c)\@call(not_implemented)
```

which states that both, line 4 and a function call to `cmp` in file `bla.c` must be covered without ever calling `not_implemented()`, is equivalent to

```
> cover /bla.c/4,/bla.c/@call(cmp)
  passing @file(bla.c)\@call(not_implemented)
```

4 Query Processing Algorithms

In this section we describe the query processing algorithms. We first outline how program source code and a query are mapped to a SAT instance, and then detail on iterative and groupwise constraint strengthening in Section 4.2.

4.1 Program Instrumentation and Interfacing with CBMC

Bounded model checkers such as CBMC reduce questions about program paths to Boolean constraints in conjunctive normal form (CNF) which are solved by standard SAT solvers. Our query solving algorithms ICS and GCS employ the functionality of CBMC to obtain SAT instances suitable for test case generation. Recall that on input of a program annotated with assertions, CBMC outputs a SAT instance whose solutions describe program paths leading to assertion violations. To make this functionality useful for test case generation, we first instrument the program with the observation automaton A such that the resulting program reaches a failing assertion in the course of an execution, iff this program execution is matched by A. We therefore implement A as a C function that *monitors* program execution. To this end, the program $\mathcal{P}$ is instrumented to contain a *logging* layer, which reports the matching predicates after each executed step to the monitor. Moreover, we inject the test goal automaton as a second monitor, which only keeps track of the states of the test goal automaton in a distinguished variable, but does not cause assertion violations. Then, using CBMC, the instrumented program is transformed into the CNF-formula $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ which is satisfied by all program executions which reach an accepting state of A within a bounded number of steps. By construction, $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ contains distinguished Boolean variables referring to the state of the query automaton Q; these variables can be used to express the individual test goals. Therefore, a constraint of the form $\phi[\pi \in \Pi_A^{\mathcal{T}}] \wedge \phi[a]$ will satisfy those program executions which (i) respect observation automaton A and (ii) satisfy test goal Ψ_a. *In the rest of this section, we will for simplicity write this constraint as* $\pi \in \Pi_A^{\mathcal{T}} \ \wedge \ \pi \models \Psi_a$*, and tacitly assume the translation to CBMC described above.*

4.2 Guided SAT Enumeration

To generate a test suite Γ for a transition system $\mathcal{T}$ matching the query $\langle A, Q \rangle$, i.e., to achieve $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$, we introduce *iterative constraint strengthening (ICS)*. In ICS,

we build a test suite Γ iteratively from a sequence of test suites $\Gamma_0 \subset \Gamma_1 \subset \cdots \subset \Gamma_m$ with $\Gamma_0 = \emptyset$ and $\Gamma_q = \{\pi_1, \ldots, \pi_q\}$ for $1 \leq q \leq m$. In the m-th iteration, we reach a fixpoint when no more new goals can be covered.

Algorithm Overview. In the q-th iteration we build the *path constraint* ICSPC_q (Equation (1)) and obtain the test case π_{q+1} as one of its solutions. Here, ICSPC_q describes those paths in $\Pi_A^{\mathcal{T}}$ which cover a hitherto uncovered test goal. If no such test goal exists any more, ICSPC_q becomes unsatisfiable. Having determined a new test case π_{q+1}, we build ICSPC_{q+1} and continue the procedure with the $(q+1)$-st iteration until we reach an iteration m where ICSPC_m becomes unsatisfiable.

In order to fit the framework of *incremental SAT solving* (cf. [12]), we rewrite ICSPC_q (Equation (2)) in such a way that we are able to describe ICSPC_{q+1} incrementally in terms of ICSPC_q by *only adding* new constraints *without removing or changing* previously added constraints (Equation (3)). Using this incremental formulation of ICSPC_q, we describe iterative constraint strengthening (ICS) based upon an incremental SAT solver in Listing 2. The m paths finally collected by ICS constitute indeed a covering test suite (Theorem 1).

Path Constraints. The initial path constraint ICSPC_0 requires that a path is in $\Pi_A^{\mathcal{T}}$ and covers at least one of the test goals Ψ_a for $a \in \mathsf{acc}(Q)$. Subsequently, in ICSPC_q, we require the path to cover at least one test goal Ψ_a which remained *uncovered* by the test suite Γ_q. Since Γ_{q+1} must cover at least one more test goal than Γ_q, it suffices to *strengthen* the constraint ICSPC_q to obtain ICSPC_{q+1}. Below, we write $\mathsf{uncov}_q = \{a \in \mathsf{acc}(Q) \mid \Gamma_q \not\models \Psi_a\}$ for the set of accepting states which correspond to test goals not covered in Γ_q. Note that $\mathsf{uncov}_0 = \mathsf{acc}(Q)$ since $\Gamma_0 = \emptyset$ covers no test goals at all. Then, for $0 \leq q \leq m$, we search for a solution π_{q+1} to the q-th constraint

$$\mathsf{ICSPC}_q(\pi) := \pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a \tag{1}$$

Note that the empty disjunction is equivalent to false, i.e., if $\mathsf{uncov}_q = \emptyset$, then $\mathsf{ICSPC}_q \equiv \mathsf{false}$. Thus, ICSPC_q is satisfied by exactly those paths in $\Pi_A^{\mathcal{T}}$ which satisfy at least one *feasible* test goal still *uncovered* by Γ_q. If no such test goal exists, i.e., if Γ_q *achieves coverage,* then ICSPC_q is unsatisfiable.

Incremental Path Constraints. In incremental SAT solving, we use a single persistent clause database for consecutive solver invocations. When the SAT solver finds a solution, we add new clauses to the clause database, but do not remove any clauses. When the execution of the SAT solver is continued, the learned clauses obtained during earlier invocations remain valid and help to guide the search of the solver. Therefore, we have to construct ICSPC_{q+1} from ICSPC_q by only adding further constraints to the clause database. Observe that $\mathsf{uncov}_{q+1} \subset \mathsf{uncov}_q$ holds for $0 \leq q \leq m-1$. Thus in going from ICSPC_q to ICSPC_{q+1}, we have to remove all test goals Ψ_a with $a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ from the disjunction $\bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a$ occurring in Equation (1). To do so, we

introduce a new Boolean variable S_a for each accepting state $a \in \mathsf{acc}(Q)$ and write ICSPC_q equisatisfiable as

$$\mathsf{ICSPC}_q(\pi) := \left[\pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a \in \mathsf{acc}(Q)} (S_a \wedge \pi \models \Psi_a) \right] \wedge \bigwedge_{a \notin \mathsf{uncov}_q} \neg S_a \tag{2}$$

Thus ICSPC_q consists of (a) an initial expression, shown above in square brackets, which remains unchanged throughout all iterations, and (b) a conjunction which is expanded from one iteration to the next. Adding $\neg S_a$ to the constraint renders the corresponding disjunct $S_a \wedge \pi \models \Psi_a$ unsatisfiable, and therefore only the disjuncts for $a \in \mathsf{uncov}_q$ remain enabled. Note that for ICSPC_0 we have $\mathsf{true} \equiv \bigwedge_{a \notin \mathsf{uncov}_0} \neg S_a$. Thus, in each iteration step, we use

$$\mathsf{ICSPC}_{q+1}(\pi) := \mathsf{ICSPC}_q(\pi) \wedge \bigwedge_{a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}} \neg S_a \tag{3}$$

to obtain ICSPC_{q+1} from ICSPC_q. Since we only add further constraints conjunctively, this approach fits the requirements of incremental SAT solving.

Iterative Constraint Strengthening. In our presentation of the algorithm, we assume a SAT solver which supports the following methods: (a) *Adding constraints* with add(ϕ): The method takes an arbitrary constraint ϕ over variables from arbitrary finite domains. While we use such a general interface to simplify the presentation of our algorithm, our implementation is based upon the SAT instance $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ which we described in Section 4.1. (b) *Checking for satisfiability* with satisfiable(): The method returns true iff there exists a solution to the constraints added to the clause database so far. If a call to satisfiable() returns true, a witness is cached. (c) *Obtaining a solution* with solution(): The method returns the last witness cached in a call to satisfiable().

Listing 2. Iterative ConstraintStrengthening (ICS)

```
1  func ICS(Π_A^T, ⟨A, Q⟩)
2  begin
3    q := 0; Γ_0 := ∅; uncov_0 := acc(Q);
4    add(π ∈ Π_A^T ∧ ⋁_{a∈acc(Q)} (S_a ∧ π ⊨ Ψ_a));
5    while satisfiable () do begin
6      π_{q+1} := solution();
7      Γ_{q+1} := Γ_q ∪ {π_{q+1}}; uncov_{q+1} := ∅;
8      forall a ∈ uncov_q do
9        if π_{q+1} ⊨ Ψ_a then add(¬S_a);
10       else uncov_{q+1} := uncov_{q+1} ∪ {a};
11     q := q + 1;
12   end;
13   return Γ_q;
14 end;
```

The resulting procedure ICS is shown in Listing 2. In line 3 we initialize the iteration counter q, the first test suite Γ_0, and the set of test goals uncov_0 uncovered by Γ_0. Then in line 4, we add the initial expression from Equation (2) and start the search for the first solution in line 5. If a solution is found, it is obtained from the solver, assigned to π_{q+1}, and added to Γ_{q+1}. Then, after initializing uncov_{q+1}, we update the clause database following Equation (3) and fill the set uncov_{q+1} in lines 8 to 10: For each yet uncovered state $a \in \mathsf{uncov}_q$, we check whether π_{q+1} satisfies Ψ_a. If this is the case, $a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ holds, and thus we add $\neg S_a$ in line 9. Otherwise a remains

uncovered by Γ_{q+1} and hence we add a to uncov_{q+1} in line 10. Once no further solution is found in line 5, the accumulated suite Γ_q is returned.

Theorem 1 (Correctness of Iterative Constraint Strengthening). *The test suite* Γ *returned by the algorithm* $\mathrm{ICS}(\Pi_A^{\mathcal{T}},\langle A,Q\rangle)$ in Listing 2 satisfies $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$.

Remark 3 (Nondeterminism in Choosing π_{q+1}). Our algorithm leaves the particular choice of π_{q+1} open to the underlying SAT solver (line 6). Potential optimizations could control this choice to minimize the number of test cases necessary to obtain coverage.

Groupwise Constraint Strengthening. Certain regular coverage criteria, such as predicate complete or multiple condition coverage, require an *exponential number of test goals.* For example, recall that multiple condition coverage (Section 3.1) has one test goal for each basic block and *each possible evaluation of all conditions* involved in deciding which edge to choose in leaving the basic block. Hence, the number of test goals is exponential in the number of conditions in each decision. For this reason, the disjunction in ICSPC_0 will be of exponential size—thus rendering iterative constraint strengthening hard for such coverage criteria.

To mitigate this situation, we introduce *groupwise constraint strengthening (GCS)* as an optimization of iterative constraint strengthening. GCS can be combined with ICS and allows to handle all test goals which are state predicates. Let us thus for simplicity assume that all test goals Ψ_a for $a \in \mathsf{acc}(Q)$ are state predicates. To apply GCS, we require the test goals to be partitioned into k distinct *groups* $G_i = \{\Psi_i^1,\ldots,\Psi_i^{k_i}\}$ *of mutually exclusive test goals* for $1 \le i \le k$, i.e., we require that there exists no *state* s with $s \models \Psi_i^g$ and $s \models \Psi_i^h$ for all $1 \le g \ne h \le k_i$ and $1 \le i \le k$.

In the GCS algorithm, we avoid the construction of the initial and very large disjunction $\bigvee_{a\in\mathsf{uncov}_q} \pi \models \Psi_a$, as it appears in ICSPC_q (Equations (1) and (2)): Instead of individual test goals, we use a small number of *compound test goals* comp_i, where each compound test goal represents the goals of the whole group $G_i = \{\Psi_i^1,\ldots,\Psi_i^{k_i}\}$ of individual test goals Ψ_i^j. To represent group G_i, its compound test goal comp_i has to be semantically equivalent (but usually not identical) to $\bigvee_{j=1}^{k_i} \Psi_i^j$. It is important to note however that in many practical cases, comp_i can be *formulated much more succinctly* than $\bigvee_{j=1}^{k_i} \Psi_i^j$. For example, in case of multiple condition coverage, we partition the goals into groups according to the blocks they relate to. Then, $s \models \mathsf{comp}_i$ holds for a state s iff s visits the i-th basic block, i.e., comp_i has the form $\mathsf{pc} = \mathrm{const}$.

Starting with the compound test goal comp_i, we add for each covered test goal Ψ_i^j of group G_i, i.e., for each $\Psi_i^j \in G_i \setminus \mathsf{uncov}_q$, its negation $\neg\Psi_i^j$ to the corresponding compound test goal. This approach yields for each group G_i an *aggregate test goal*

$$\mathsf{aggr}_i^q := \mathsf{comp}_i \wedge \bigwedge_{\Psi_i^j \in G_i \setminus \mathsf{uncov}_q} \neg\Psi_i^j \tag{4}$$

Since we use aggr_i^q to represent the remaining uncovered test goals $G_i \cap \mathsf{uncov}_q$ of the group G_i in iteration q, we will rely on the equivalence

$$\mathsf{aggr}_i^q \equiv \bigvee_{\Psi_i^j \in G_i \cap \mathsf{uncov}_q} \Psi_i^j \tag{5}$$

which follows from the construction and the mutual exclusiveness of the test goals within each group G_i. Written in the form of Equation (5), aggr_i^q does not explicitly refer to any infeasible test goals and only involves *feasible* test goals as subexpressions. This significantly reduces the size of the constructed constraint.

Having defined aggr_i^q in this way, GCS proceeds like ICS but with Equation (1) replaced by

$$\mathsf{GCSPC}_q(\pi) := \pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{i=1}^{k} \pi \models \mathsf{aggr}_i^q \tag{6}$$

Similar to ICS we also adopt GCSPC_q to fit incremental SAT solving: More precisely, we leave the overall constraint (Equation (6)) unchanged and replace aggr_i^q (Equation (4)) by an equisatisfiable and incrementally expandable expression. Thus, we can incrementally strengthen aggr_i^q for each group individually.

The effectiveness of GCS as an optimization of ICS relies on three conditions: (a) The overall number of groups must be small, since we maintain for each group G_i a constraint aggr_i^q. (b) The compound test goal comp_i must be available in a succinct formulation. (c) The fraction of *feasible* test goals Ψ_i^j in each group G_i must be small, since the negation of each feasible test goal is added to aggr_i^q in some iteration q. Conditions (a) and (b) hold for important coverage criteria such as multiple decision or predicates complete coverage. If condition (c) does not hold, then the number of required test cases will be large – but this is inherent in the coverage criterion and not an artefact of GCS.

Remark 4 (Mutual Exclusiveness: State vs. Path Predicates). It is tempting to assume that the mutual exclusiveness defined in terms of states is easily generalized to the level of path predicates. However, this is not the case as mutually exclusive state predicates *do not result* in mutually exclusive path predicates because of their implicit existential quantification, cf. Definition 3 and Remark 1.

5 Experimental Results

In our experiments we investigated test case generation for basic block (BB) and condition coverage (CC). We performed our experiments on a 3.0 GHz AMD64 system with 8 GB RAM. The table below summarizes our results with respect to BLAST. The column "Min" shows the number of test cases removed by our test suite minimization algorithm. Our current implementation of FSHELL is an optimized version of that presented in [4]. It generates fewer test cases, and, after test case generation for basic block coverage, FSHELL minimizes an obtained test suite. The results for BLAST are taken literally from [3], because the version of BLAST performing test case generation is currently unavailable. Beyer et al. performed their experiments on a 3.06 GHz Dell Precision 650 with 4 GB RAM. FSHELL outperforms BLAST, as we achieve coverage with fewer test cases faster. Besides the experiments on the device drivers from BLAST we conducted experiments on an engine controller (`matlab.c`) provided by an industrial collaborator from the automotive industries. It is generated from a MATLAB/Simulink model. Furthermore, we ran our tool on preprocessed sources (`autopilot.i`) generated from

Source file	LLOC	BLAST (BB) #cases	BLAST (BB) Time[s]	BB #cases	BB Time[s]	BB Min*	CC #cases	CC Time[s]
kbfiltr.i	4879	39	300	26	18	6	98	24
floppy.i	6435	111	1500	63	1041	10	175	1259
cdaudio.i	8022	85	1500	71	1240	7	161	1243
parport.i	20698	213	5460	134	1859	21	351	2915
parclass.i	45283	219	2520	156	1324	16	392	2070
matlab.c	2033	-	-	5	30	1	16	31
autopilot.i	3141	-	-	206	894	14	450	1358

source code in PapaBench[2]. The results show that FSHELL scales well when moving from basic block coverage to condition coverage. Experiments concerning more sources and more complex queries can be found in [11].

6 Conclusion

In this paper, we introduced a query language for test case specification together with a query solving backend based on bounded model checking. Our backend is based on two new algorithms which guide the SAT solver to efficiently enumerate a test suite. Our implementation FSHELL demonstrates the effectiveness and versatility of our approach.

References

1. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 1–22. Springer, Heidelberg (2005)
2. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE, pp. 326–335 (2004)
4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 209–213. Springer, Heidelberg (2008)
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
6. 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)
7. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 327–341. Springer, Heidelberg (2002)

[2] http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97

8. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In: IRI, pp. 493–498 (2004)
9. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA, pp. 97–107 (2004)
10. Hamon, G., de Moura, L.M., Rushby, J.M.: Generating Efficient Test Sets with a Model Checker. In: SEFM, pp. 261–270 (2004)
11. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. Technical Report TUD-CS-2008-1013, TU Darmstadt (2008)
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

Average-Price-per-Reward Games on Hybrid Automata with Strong Resets*

Marcin Jurdziński, Ranko Lazić, and Michał Rutkowski

Department of Computer Science, University of Warwick, UK

Abstract. We study price-per-reward games on hybrid automata with strong resets. They generalise priced games previously studied and have applications in scheduling. We obtain decidability results by a translation to a novel class of finite graphs with price and reward information, and games assigned to edges. The cost and reward of following an edge are determined by the outcome of the edge game that is assigned to it.

1 Introduction

Hybrid Systems and Automata. Systems that exhibit both discrete and continuous behaviour are referred to as *hybrid systems*. Continuous changes to the system's state are interleaved with discrete ones, which may alter the constraints for future continuous behaviours. *Hybrid automata* are a formalism for modeling hybrid systems [1]. Hybrid automata are finite automata augmented with continuous real-valued variables. The discrete states can be seen as modes of execution, and the continuous changes of the variables as the evolution of the system's state over time. The mode specifies the continuous dynamics of the system, and mode changes are triggered by the changes in variable's values.

Reachability [2,3,4,5] and optimal reachability [6,7] analysis for hybrid automata have been studied. In [6,8] the optimality of infinite behaviours is also addressed.

Optimal Schedule Synthesis. Hybrid systems have been successfully applied to modeling scheduling problems [9]. In this setting, an execution of the automaton is a potential schedule. In [8], the authors equip timed automata, a subclass of hybrid systems, with price and reward information. Each schedule comes at a price, but provides a certain reward. The price-over-reward ratio can be seen as a measure of how cost-effective the schedule is. A natural example of a reward is time. In this case, price-per-time unit is being optimised. The problem that arises is the synthesis of an optimal schedule, i.e., a schedule that minimises the price-over-reward ratio. Reachability-price-per-reward analysis is used in the synthesis of finite optimal schedules. When dealing with reactive behaviour, optimality of infinite schedules becomes more important. Average-price-per-reward analysis, where the average price-over-reward ratio of a single step in the execution is optimised, is used in this context [8].

* This research was supported in part by EPSRC project EP/E022030/1.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 167–181, 2009.

We follow this direction and study the problem in the context of hybrid automata with strong resets. Our research shares the same motivation, but both the model and the techniques used differ. In [8] timed automata, a different class of hybrid automata, is considered, and an abstraction technique, known as "corner-point abstraction", is used. We, on the other hand, use an abstraction, that was first proposed in [6], to reduce to price-reward graphs, that are introduced in this paper.

Controller Synthesis. The designer of a system often lacks full control over its operation. The behaviour of the system is a result of an interaction between a controller and the environment. This gives rise to the *controller synthesis* problem (first posed by Church [10]), where the goal is to design a program such that, regardless of the the environment's behaviour, the system behaves correctly and optimally. A game-based approach was proposed in [11], and was applied to hybrid automata [12,13] and timed automata [14]. There are two players, *controller* and *environment*, and they are playing a zero-sum game. The game is played on the hybrid automaton and consists of rounds. As usual, we use player Min to denote the controller and player Max to denote the environment. In each round, Min proposes a transition. In accordance with the game protocol, Max can choose to perform this or another transition.

Determinacy and decidability are important properties of zero-sum games. A determined zero-sum game has a value, and admits almost-optimal controllers (strategies). A determined game is decidable if, given some some rational number, we can decide whether the value of the game is greater than the number.

Hybrid Games with Strong Resets. We are considering a subclass of hybrid automata: hybrid automata with strong resets (HASR). In order to represent the automaton finitely, we require that all the components of the system are first-order definable over the ordered field of reals. The term "strong resets" comes from the property of the system that all continuous variables are nondeterministically reset after each discrete transition. As opposed to timed automata, where flow rates are constant, and reseting of the variables upon a discrete transition is not compulsory [2], HASR allow for rich continuous dynamics [4,12,13].

In the game setting, we allow only for alternating sequences of timed and discrete transitions [12,13]. A timed transition followed by a discrete one will be called a timed action. Allowing an arbitrary number of continuous transitions prior to a discrete one, without the requirement of o-minimality, renders it impossible to construct a bisimulation of finite index [15,16].

Contributions. We are considering *average-price-per-reward games*, where players are trying to optimise the average price-over-reward ratio of a timed action. Our main result is that average-price-per-reward games are determined and decidable. It is obtained through a reduction to games on finite price-reward graphs (PRGs) introduced in this paper.

To reduce hybrid average-price-per-reward games to their counterparts on PRGs we use the same equivalence as in [6]. However, there are two significant

contributions with respect to [6]. Firstly, we are considering the average price-over-reward ratio, whereas only average price per transition was considered in [6]. The first is significantly more complex. Secondly, we introduce a novel class of finite graphs with price and reward information, and games assigned to edges (PRG). In this paper we show that average-price-per-reward games on PRGs are determined and decidable.

We believe that our results and technical developments concerning PRGs are interesting in their own right. To characterise game values we use a technique, referred to as optimality equations [6,14]. What is novel is that we use the values of edge games to express optimality criteria in these equations. The proof that solutions to the optimality-equations exist (and hence the games are determined) relies on the properties of the equations, not of a particular game (on a PRG). This makes us believe that our technique is robust, and can be used to solve related games such as, reachability-price-per-reward. To show determinacy and decidability we only need to express optimality criteria, for a given game on a PRG, in terms of edge games' values.

It is worth noting that our results can be easily extended to *relaxed hybrid automata* [5], where the strong reset requirement is replaced by a requirement that every cycle in the control graph has a transition that resets all the variables. This extension can be achieved by a refinement of the equivalence relation and a minor modification of the finite graph obtained from it. For clarity, we decided against considering this more general model.

Organisation. Sec. 2 introduces the basic notions used throughout the paper, i.e., definability and decidability, zero-sum games, price-reward graphs, and average-price-per-reward games together with their optimality-equation characterisation. Sec. 3 contains the main technical contribution of the paper: that finite average-price-per-reward games are determined, and that almost optimal controllers exist. In Sec. 4 we state our main results: determinacy and existence of almost-optimal controllers for hybrid average-price-per-reward games.

2 Preliminaries

Here, we introduce key notions that will be used further in the paper, such as definability, decidability, and two-player zero-sum games on price-reward graphs. In Sec. 2.3, we introduce average-price-per-reward games, and optimality equations as means of characterisation (Thm. 4).

Throughout the paper, $\mathbb{R}_\infty$ denotes the set of real numbers augmented with positive and negative infinities, and $\mathbb{R}_+$ and $\mathbb{R}_\oplus$ denote the sets of positive and non-negative reals, respectively. If $G = (V, E)$ is a graph, then for a vertex v, we write vE to denote the set $\{v' : (v, v') \in E\}$ of its successors.

2.1 Definability and Decidability

Definability. Let $\mathcal{M} = \langle \mathbb{R}, 0, 1, +, \cdot, \leqslant \rangle$ be the ordered field of reals. We say that a set $X \subseteq \mathbb{R}^n$ is *definable* in $\mathcal{M}$ if it is *first-order definable* in $\mathcal{M}$. The *first-order*

theory of $\mathcal{M}$ is the set of all first-order sentences that are true in $\mathcal{M}$. A well-known result by Tarski [17] is that the first-order theory of $\mathcal{M}$ is decidable.

Computability and Decidability. For a finite set A, we will say that $(a, x) \in A \times \mathbb{R}^n$ is rational if $x \in \mathbb{Q}^n$. Let $f : X \to \mathbb{R}$ be a partial function, that is defined on a set $D \subseteq X \subseteq \mathbb{R}^n$. We say that f is *approximately computable* if there is an algorithm that for every rational $x \in D$, and every $\varepsilon > 0$, computes a $y \in \mathbb{Q}$ such that $|y - f(x)| < \varepsilon$. It is *decidable* if the following problem is decidable: given a rational $x \in D$ and rational c, decide whether $f(x) \leqslant c$.

We extend the notions of approximate computability, and decidability to functions $f : A \times \mathbb{R}^n \to \mathbb{R}$, where A is finite, by requiring that $f(a, \cdot)$ is respectively: approximately computable, and decidable for every $a \in A$.

Proposition 1. *If a function is decidable then it is approximately computable.*

Proposition 2. *If a real partial function is definable in* $\mathcal{M}$ *then it is decidable.*

The purpose of the above definitions is to enable us to state conclusions of our definability results. By no means should they be treated as a formalisation of computation over the reals. For models of computing over the reals we refer the reader to [18,19,20].

2.2 Zero-Sum Games

In this section we introduce zero-sum games in strategic form, price-per-reward game graphs, and zero-sum price-reward games. Fundamental concepts such as: game value, determinacy, decidability, and optimal strategies are introduced in the context of games in strategic form, and are later lifted to price-reward games. Although our results concern games on price-reward game graphs, the notion of a game in strategic form will be important throughout the paper (for instance, in the formulation of the optimality equations in Sec. 2.3).

Games in Strategic Form. A zero-sum game is played by two players: Min and Max. Let Σ^{Min}, Σ^{Max} be the sets of *strategies* for players Min and Max respectively. Let $\mathcal{O}$ be the set of outputs, and let $\xi : \Sigma^{\mathrm{Min}} \times \Sigma^{\mathrm{Max}} \to \mathcal{O}$ be a function that, given strategies of players Min and Max, determines the output of the game. Finally let $\mathsf{P} : \mathcal{O} \to \mathbb{R}$, be the payoff function, which given an output determines the payoff. Player Min wants to minimise the payoff, whereas player Max wants to maximise it. A zero-sum game $\mathfrak{I}$ in a *strategic form* is given as $\langle \Sigma^{\mathrm{Min}}, \Sigma^{\mathrm{Max}}, \mathcal{O}, \xi, \mathsf{P} \rangle$. We say that $\mathfrak{I}$ is *definable* if all its components are definable. Recall that definability of a component implicitly implies that it is a subset of $\mathbb{R}^n$.

We define the lower value $\mathsf{Val}_*(\mathfrak{I}) = \sup_{\chi \in \Sigma^{\mathrm{Max}}} \inf_{\mu \in \Sigma^{\mathrm{Min}}} \mathsf{P}(\xi(\mu, \chi))$ and the upper value $\mathsf{Val}^*(\mathfrak{I}) = \inf_{\mu \in \Sigma^{\mathrm{Min}}} \sup_{\chi \in \Sigma^{\mathrm{Max}}} \mathsf{P}(\xi(\mu, \chi))$. Note that $\mathsf{Val}_*(\mathfrak{I}) \leqslant \mathsf{Val}^*(\mathfrak{I})$, and if these values are equal, then we will refer to them as the value of the game, denoted by $\mathsf{Val}(\mathfrak{I})$. We will also say that the game is *determined*. Note that, in the definitions above, we allow only pure strategies (i.e., elements of strategy sets).

For all $\mu \in \Sigma^{\mathrm{Min}}$, we define $\mathsf{Val}^{\mu}(\mathfrak{I}) = \sup_{\chi' \in \Sigma^{\mathrm{Max}}} \mathsf{P}(\xi(\mu, \chi'))$. Analogously, for $\chi \in \Sigma^{\mathrm{Max}}$ we define $\mathsf{Val}_{\chi}(\mathfrak{I}) = \inf_{\mu' \in \Sigma^{\mathrm{Min}}} \mathsf{P}(\xi(\mu', \chi))$. For $\varepsilon > 0$, we say

that $\mu \in \Sigma^{\mathrm{Min}}$ is *ε-optimal* if we have that $\mathsf{Val}^{\mu}(\mathcal{D}) \leqslant \mathsf{Val}^{*}(\mathcal{D}) + \varepsilon$. We define ε-optimality of strategies for Max analogously.

There are cases in which the desired payoff function is only partially defined on the set of outputs. To remedy this, *lower* $\mathsf{P}_{*} : \mathcal{O} \to \mathbb{R}$ and *upper* $\mathsf{P}^{*} : \mathcal{O} \to \mathbb{R}$ payoff functions are used. It is required that $\mathsf{P}_{*} \leqslant \mathsf{P}^{*}$. Due to this generalisation, the lower value, and the value of player Max's strategy are defined using the lower payoff, whereas the analogous definitions for the upper value and the value of player Min's strategy use the upper payoff.

Price-Reward Game Graphs. Let $\langle \mathsf{S}, \mathsf{E} \rangle$ be a directed graph and let $\mathsf{S}^{\mathrm{Min}} \uplus \mathsf{S}^{\mathrm{Max}}$ be a partition of S. Let $\mathcal{I}$ be the set of inputs, and let $\Theta^{\mathrm{Min}}, \Theta^{\mathrm{Max}} : \mathsf{E} \to 2^{\mathcal{I}}$ be functions that to every edge, assign the sets of valid inputs. Finally, let $\pi : \mathsf{E} \times \mathcal{I}^2 \to \mathbb{R}$ be a price function, and $\kappa : \mathsf{E} \times \mathcal{I}^2 \to \mathbb{R}_{\oplus}$ be a reward function. A price-reward game graph Γ is given as a tuple $\langle \mathsf{S}^{\mathrm{Min}}, \mathsf{S}^{\mathrm{Max}}, \mathsf{E}, \mathcal{I}, \Theta^{\mathrm{Min}}, \Theta^{\mathrm{Max}}, \pi, \kappa \rangle$. It is said to be *definable* if all its components are definable. When the payoff functions are given, we will refer to Γ as a *price-reward game*.

Intuitively the game is played by moving a token, along the edges, from one state to another. The states are partitioned between players Min and Max. The owner of the state decides along which edge to move the token. The price (reward) of an edge depends on the supplied inputs, one of each is chosen by Min and the other one by Max. The game is played indefinitely. A payoff function determines how the prices (rewards) of individual moves contribute to the overall value of a particular play. The players Min and Max are trying to minimise and maximise (respectively) the value of the payoff function.

We write $s \to_{\theta} s'$ to denote a move, where $e = (s, s') \in \mathsf{E}$ and $\theta \in \Theta^{\mathrm{Min}}(e) \times \Theta^{\mathrm{Max}}(e)$. The price of the move is $\pi(e, \theta)$ and the reward is $\kappa(e, \theta)$. A run is a (possibly infinite) sequence of moves $\rho = s_0 \to_{\theta_1} s_1 \to_{\theta_2} s_2 \cdots$. The set of all valid runs of Γ is denoted by Runs, and its subset of all valid *finite runs* by $\mathrm{Runs}_{\mathrm{fin}}$.

A *state strategy* of player Min is a partial function $\mu^{\mathsf{S}} : \mathrm{Runs}_{\mathrm{fin}} \to \mathsf{E}$ which is defined on all runs ending in $s \in \mathsf{S}^{\mathrm{Min}}$. A strategy is called positional if it can be viewed as a function $\mu^{\mathsf{S}} : \mathsf{S}^{\mathrm{Min}} \to \mathsf{E}$. Given an edge e, an e-strategy of player Min is an element $x \in \Theta^{\mathrm{Min}}(e)$. An *edge strategy* μ^{E} of player Min is a function, that to every edge e assigns an e-strategy.

A strategy μ of player Min is a pair $(\mu^{\mathsf{S}}, \mu^{\mathsf{E}})$ of state and edge strategies. We denote the set of all strategies by Σ^{Min}. We say that μ is positional if μ^{S} is positional. We denote the set of all positional strategies by Π^{Min}. Strategies of player Max are defined analogously.

Given strategies μ and χ of players Min and Max and some state s, we write $\mathsf{Run}(s, \chi, \mu)$ to denote the run starting in s resulting from players playing according to their strategies μ and χ.

Determinacy. Let $\mathsf{P}_{*} : \mathrm{Runs} \to \mathbb{R}$ and $\mathsf{P}^{*} : \mathrm{Runs} \to \mathbb{R}$ be the upper and lower payoff functions. Typically, payoff functions are expressions involving prices and rewards of individual transitions.

Given a state s, let $\mathcal{D}_s = \langle \Sigma^{\mathrm{Min}}, \Sigma^{\mathrm{Max}}, \mathrm{Runs}, \mathsf{Run}(s, \cdot, \cdot), \mathsf{P}^{*}, \mathsf{P}_{*} \rangle$. We say that the game Γ is *determined* from s if $\mathsf{Val}_{*}(\mathcal{D}_s) = \mathsf{Val}^{*}(\mathcal{D}_s)$, and *positionally deter-*

mined if $\mathsf{Val}(\Game_s) = \inf_{\mu \in \Pi^{\mathrm{Min}}} \mathsf{Val}^{\mu}(\Game_s) = \sup_{\chi \in \Pi^{\mathrm{Max}}} \mathsf{Val}_{\chi}(\Game_s)$. We say that $\Game$ is determined if it is determined from every state.

For simplicity we will write $\mathsf{Val}(s)$ rather then $\mathsf{Val}(\Game_s)$, in the context of price-reward games, so Val can be viewed as a partial function $\mathsf{S} \to \mathbb{R}$.

Decidability. We will say that a price-reward game $\Game$ is *decidable* if the partial function $\mathsf{Val} : \mathsf{S} \to \mathbb{R}$ is decidable. We emphasise that Val is a partial function because $\Game$ does not have to be determined from every state.

2.3 Average-Price-per-Reward Games

In this section, we introduce average-price-per-reward games, and provide a characterisation of game values using a set of equations, referred to as optimality equations. The key result is Thm. 4, which states that solutions to optimality equations coincide with game values.

The results presented here are general, and will be applied to finite average-price-per-reward games (Sec. 3) as well as to their hybrid counterparts (Sec. 4). The fact that, in both cases, the game values are characterised using optimality equations will be used in the proof of the reduction from hybrid games to finite games (Sec. 4). Notions and arguments similar to those introduced here have been used in the past [6,14]. We decided to state them in full detail, because they form an important part of our reasoning and provide valuable insight.

The goal of player Min in the *average-price-per-reward game* $\Game$ is to minimise the average price-over-reward ratio in a run, and the goal of player Max is to maximise it. We define the upper and lower payoff functions in the following way:

$$\mathsf{P}^*(\rho) = \limsup_{n\to\infty} \frac{\sum_{i=0}^{n} \pi(e_{i+1}, \theta_{i+1})}{\sum_{i=0}^{n} \kappa(e_{i+1}, \theta_{i+1})} \quad \text{and} \quad \mathsf{P}_*(\rho) = \liminf_{n\to\infty} \frac{\sum_{i=0}^{n} \pi(e_{i+1}, \theta_{i+1})}{\sum_{i=0}^{n} \kappa(e_{i+1}, \theta_{i+1})},$$

where ρ is an infinite run, $s_i \to_{\theta_{i+1}} s_{i+1}$ and $e_i = (s_i, s_{i+1})$ for all $i \geqslant 0$.

To guarantee that the payoffs, as introduced above, are always well-defined we introduce the notions of reward divergence and price-reward boundedness.

We say that $\Game$ is $\Omega(f(n))$*-reward divergent* if, for every run ρ, the function $n \mapsto \sum_{i=0}^{n} \kappa(s_i, \theta_{i+1})$ is in $\Omega(f(n))$. We assume that $\Game$ is $\Omega(n)$-reward divergent. Linear (i.e., $\Omega(n)$) reward divergence is required in the proof of Thm. 4. In the remainder of the paper $c > 0$ will be the largest number such that, for every run ρ, we have $n \mapsto \sum_{i=0}^{n} \kappa(s_i, \theta_{i+1}) \geqslant c \cdot n$.

Additionally, we require that $\Game$ is *price-reward bounded*, i.e., $|\pi| < M$ and $|\kappa| < M$ for some M. This is necessary to assure that edge games, as introduced below, are determined. Moreover, without loss of generality, we assume that the games are non-blocking, i.e., there are no sink states.

The divergence requirement can be seen as a generalisation of the non-zenoness requirement to rewards (as in [8]); we want to prevent runs that admit finite rewards. Note that if the reward is simply time, then we get the non-zenoness condition. Also note that one can can guarantee $\Omega(n)$-reward divergence by claiming that $\kappa > c$ for some $c > 0$.

Optimality Equations. Let Γ be a price-per-reward game. For every edge e, we introduce a game $\eth_e(g) = \langle \Theta^{\mathrm{Min}}(e), \Theta^{\mathrm{Max}}(e), \Theta^{\mathrm{Min}}(e) \times \Theta^{\mathrm{Max}}(e), id, \mathsf{P}_e(g)\rangle$, where g is a real-valued parameter, and $\mathsf{P}_e(g) = \pi(e) - \kappa(e) \cdot g$. We will refer to it as an *edge game*. Note that, for every $e \in \mathsf{E}$ and $g \in \mathbb{R}$, we have that $\eth_e(g)$ is determined and definable.

Let $G, B : \mathsf{S} \to \mathbb{R}$ such that the range of G is finite, and B is bounded. We say that a pair of functions (G, B) is a solution of *optimality equations* for Γ, denoted by $(G, B) \models \mathrm{Opt}(\Gamma)$, if the following conditions hold for all states $s \in \mathsf{S}^{\mathrm{Min}}$:

$$G(s) = \min_{(s,s') \in \mathsf{E}} \{G(s')\} \tag{1}$$

$$B(s) = \inf_{(s,s') \in \mathsf{E}} \{\mathsf{Val}(\eth_{(s,s')}(G(s'))) + B(s') \ : \ G(s) = G(s')\} \tag{2}$$

and if analogous two equations hold for all states in $\mathsf{S}^{\mathrm{Max}}$, with the only difference that min is substituted by max and inf by sup. The two functions G and B are called *gain* and *bias* respectively.

Remark 3. If Γ is definable then $\mathrm{Opt}(\Gamma)$ is first-order expressible in $\mathcal{M}$.

Theorem 4. *If $(G, B) \models Opt(\Gamma)$ then for every state $s \in \mathsf{S}$, the average-price-per-reward game Γ from s is determined and we have $\mathsf{Val}(s) = G(s)$. Moreover, for every $\varepsilon > 0$, positional ε-optimal strategies exist for both players.*

Corollary 5. *If there exists definable (G, B) such that $(G, B) \models Opt(\Gamma)$ and Γ is definable, then positional ε-optimal strategies are definable.*

The theorem and corollary follow from the following two lemmas and their proofs, which imply that for all states $s \in \mathsf{S}$, we have $\mathsf{Val}^*(s) \leq G(s)$ and $\mathsf{Val}_*(s) \geq G(s)$, respectively.

Lemma 6. *Let $(G, B) \models Opt(\Gamma)$. Then for all $\varepsilon > 0$, there is $\mu_\varepsilon \in \Pi_{Min}$ such that for all $\chi \in \Sigma^{Max}$ and for all $s \in \mathsf{S}$, we have $\mathsf{P}^*(\mathsf{Run}(s, \mu_\varepsilon, \chi)) \leq G(s) + \varepsilon$.*

Lemma 7. *Let $(G, B) \models Opt(\Gamma)$. Then for all $\varepsilon > 0$, there is $\chi_\varepsilon \in \Pi_{Max}$ such that for all $\mu \in \Sigma^{Min}$ and for all $s \in \mathsf{S}$, we have $\mathsf{P}_*(\mathsf{Run}(s, \mu, \chi_\varepsilon)) \geq G(s) - \varepsilon$.*

We omit the proof of Lem. 7 as it is similar to the proof of Lem. 6.

Proof. We prove Lem. 6 by observing that, for every $\varepsilon' > 0$, $g \in \mathbb{R}$, and an edge e, player Min can choose $x^e_{\varepsilon'} \in \Theta^{\mathrm{Min}}(e)$ such that $\mathsf{Val}^{x^e_{\varepsilon'}}(\eth_e(g)) \leqslant \mathsf{Val}(\eth_e(g)) + \varepsilon'$. Moreover, for every state $s \in \mathsf{S}^{\mathrm{Min}}$, player Min can choose an edge $e = (s, s')$ such that:

$$G(s) = G(s')$$
$$B(s) \geqslant \mathsf{P}_e(G(s'))(x^e_\varepsilon, y) + B(s') - \varepsilon', \text{ for all } y \in \Theta^{\mathrm{Max}}(e).$$

We will call this choice, of an edge and an edge strategy, ε'-optimal. It remains to show that if, in every $s \in \mathsf{S}^{\mathrm{Min}}$, $\mu_\varepsilon(s)$ is a $(c \cdot \varepsilon)$-optimal choice, then μ_ε is ε-optimal.

Let $\varepsilon > 0$ and $\mu_\varepsilon \in \Pi^{\mathrm{Min}}$ be ε-optimal for every state, and let $\chi \in \Sigma^{\mathrm{Max}}$ be arbitrarily chosen. If $s_i \to_{\theta_{i+1}} s_{i+1}$ is the $i+1$-th step of $\mathsf{Run}(s, \mu_\varepsilon, \chi)$, then $G(s_i) \geqslant G(s_{i+1})$. The range of G is finite, hence there is $K \in \mathbb{N}$ such that, for all $i \geqslant K$, $G(s_i) = g$, where $g = G(s_K)$.

Let $N \geqslant K$. For $i = K, \ldots, N$, the following holds, $B(s_i) \geqslant \mathsf{P}_e(g)(\theta_{i+1}) + B(s_{i+1}) - c \cdot \varepsilon$. If we sum up the $N - K + 1$ inequalities $(\mathsf{P}_e(g)(\theta) = \pi(e, \theta) - \kappa(e, g) \cdot g)$, we get:

$$\sum_{i=K}^{N-1} B(s_i) \geqslant \sum_{i=K+1}^{N} \pi(e_i, \theta_i) - g \cdot \sum_{i=l+1}^{k} \kappa(e_i, \theta_i) + \sum_{i=K+1}^{N} B(s_i) - (N - K + 1) \cdot c \cdot \varepsilon$$

That simplifies to:

$$\frac{B(s_K) - B(s_N)}{\sum_{i=K+1}^{N} \kappa(e_i, \theta_i)} + g \geqslant \frac{\sum_{i=K+1}^{N} \pi(e_i, \theta_i) - (N - K + 1) \cdot c \cdot \varepsilon}{\sum_{i=K+1}^{N} \kappa(e_i, \theta_i)} \geqslant \mathsf{P}^*(\mathsf{Run}(s, \mu_\varepsilon, \chi)) - \varepsilon$$

Recall that B is bounded, and that Γ is $\Omega(n)$-reward divergent with a constant c (which implies that $((N - K + 1) \cdot c \cdot \varepsilon) / \sum_{i=K+1}^{N} \kappa(e_i, \theta_i) \leqslant \varepsilon$). This yields the desired result. □

3 Finite Average-Price-per-Reward Games

In this section we state (Thm. 8) and prove (Cor. 13) our technical results, i.e., that finite average-price-per-reward games are determined and decidable[1].

To guarantee uniqueness of the constructions, and for technical convenience, we fix a linear order on the states of the game graph. Given a subgraph $S \subseteq \Gamma$, $\min(S)$ denotes the smallest state in S.

Theorem 8. *Finite average-price-per-reward games are positionally determined and decidable.*

We prove the theorem using the optimality-equation characterisation from Sec. 2.3, and by showing that, in the case of finite price-reward graphs, solutions to optimality equations exist.

Note that we can apply the results from Sec. 2.3 to finite graphs, because gain and bias always have finite ranges.

Strategy Subgraphs. Let Γ be a price-reward game graph. Let μ^{S} be a positional state strategy for player Min. Such a strategy induces a subgraph of Γ, where the

[1] By finite we mean, that the directed graph $\langle \mathsf{S}, \mathsf{E} \rangle$ is finite.

E relation is substituted by E_μ defined as $E_\mu = \{(s, s') : s \in \mathsf{S}^{\mathrm{Min}} \text{ and } \mu^{\mathsf{E}}(s) = s', \text{ or } s \in \mathsf{S}^{\mathrm{Max}}\}$. We denote this game graph by $\Gamma_{\mu^{\mathsf{S}}}$.

A finite connected price-reward game graph of out-degree one is called a *sun*. Such a graph contains a unique cycle, referred to as the *rim*. States which are on the rim are called *rim states* and the remaining ones are called *ray states*.

Remark 9. If $\mu \in \Pi^{\mathrm{Min}}$, $\chi \in \Pi^{\mathrm{Max}}$, and Γ is a price-reward game graph, then $\Gamma_{\mu^{\mathsf{S}}\chi^{\mathsf{S}}}$ is a set of suns.

Game Graphs of Out-Degree One. In price-reward game graphs of out-degree one, strategies of both players are reduced to edge-strategies only. Without loss of generality, we can assume that the price-reward game Γ is defined on a single sun. We now provide a characterisation of upper and lower game values using the values of the rim edge games.

Lemma 10. *Let Γ be a price-reward game defined on a sun, and let $e_1, \ldots, e_k$ denote the edges that form the rim of that sun. Given a parameter $p \in \mathbb{R}$, the following is true for every state s:*

- *If $\sum_{i=1}^{k} \mathsf{Val}(\eth_{e_i}(p)) \geqslant 0$ then $p \leqslant \mathsf{Val}^*(s)$,*
- *If $\sum_{i=1}^{k} \mathsf{Val}(\eth_{e_i}(p)) \leqslant 0$ then $p \geqslant \mathsf{Val}_*(s)$.*

Strict inequalities on the left hand side imply strict inequalities on the right hand side.

Proof. The proof is similar to that of Lemmas 6 and 7. We only sketch the proof of the first statement, as the other is symmetric.

Let χ be a strategy of player Max such that it is $c \cdot \varepsilon$-optimal for every edge game $\eth_{e_i}(p)$, for some $\varepsilon > 0$ and $i = 1, \ldots, k$. If μ is a strategy of player Min, then for every edge e_i:

$$\pi(e_i, \chi(e_i), \mu(e_i)) - \kappa(e_i, \chi(e_i), \mu(e_i)) \cdot p + \varepsilon \geqslant \mathsf{Val}(\eth_{e_i}(p))$$

if we add up the k inequalities we get:

$$\sum_{i=1}^{k} \pi(e_i, \chi(e_i), \mu(e_i)) - \sum_{i=1}^{k} \kappa(e_i, \chi(e_i), \mu(e_i)) \cdot p + k \cdot c \cdot \varepsilon \geqslant 0$$

which gives:

$$\frac{\sum_{i=1}^{k} \pi(e_i, \chi(e_i), \mu(e_i))}{\sum_{i=1}^{k} \kappa(e_i, \chi(e_i), \mu(e_i))} + \varepsilon \geqslant p$$

This, due to the arbitrary choice of ε and μ, finishes the proof. □

Theorem 11. *Solutions to optimality equations for average-price-per-reward games on graphs of out-degree one exist.*

Proof. a finite average-price-per-reward game on a graph of out-degree one, and let S be one of the suns. For every state, both the upper and lower values are finite (recall that Γ is price-reward bounded and linearly reward divergent). Using binary search, together with Lem. 10, it follows that they are indeed equal.

Let g be the value of the game on sun S. We set the gain of all states to g, and the bias of $\min(S)$ to zero. The bias of the remaining states is set to the weight of the shortest path to $\min(S)$, assuming $\mathsf{Val}(\partial_e(g))$ to be the weight on the edge e. Gain and bias functions defined this way satisfy optimality equations. □

General Case. We have proved that games on graphs of out-degree one are determined. We will now use this result to prove determinacy in the general case.

Let μ^{S} and χ^{S} be state strategies for players Min and Max respectively, and let (G, B) be gain and bias functions such that $(G, B) \models \mathrm{Opt}(\Gamma_{\mu^{\mathsf{S}}\chi^{\mathsf{S}}})$. Given $s \in \mathsf{S}^{\mathrm{Min}}$ and $e = (s, s') \in \mathsf{E} \setminus \mathsf{E}_{\mu^{\mathsf{S}}\chi^{\mathsf{S}}}$, we say that e is an *improvement* of μ^{S}, with respect to χ^{S}, if $G(s) > G(s')$, or $G(s) = G(s')$ and $B(s) > \mathsf{Val}(\partial_e(G(s)) + B(s')$. A strategy μ'^{S} is an improvement of μ^{S} with respect to χ^{S} if for every state s, either $\mu^{\mathsf{S}}(s) = \mu'^{\mathsf{S}}(s)$, or $\mu'^{\mathsf{S}}(s) = s'$ and (s, s') is an improvement of μ^{S} with respect to χ^{S}. An improvement is strict if $\mu^{\mathsf{S}} \neq \mu'^{\mathsf{S}}$. An improvement of χ^{S} is defined similarly.

We say that χ^{S}, a state strategy for player Max, is a *best response* to μ^{S}, a state strategy of player Min, if there are no possible improvements of χ^{S} with respect to μ^{S}.

To prove the existence of best response strategies we apply Thm. 12 and the fact that the set of edge strategies is finite, to average-price-per-reward games, in which all the states belong to only one player.

Theorem 12. *Let μ^{S} be a state strategy of player Min, χ^{S} a best response strategy of player Max, and (G, B) gain and bias such that $(G, B) \models \mathit{Opt}(\Gamma_{\mu^{\mathsf{S}}\chi^{\mathsf{S}}})$. If μ'^{S} is an improvement of μ^{S} with respect to χ^{S}, χ'^{S} is a best response to μ'^{S}, and $(G', B') \models \mathit{Opt}(\Gamma_{\mu'^{\mathsf{S}}\chi'^{\mathsf{S}}})$, then the following holds for every state s:*

1. *$G(s) < G'(s)$, or*
2. *$G(s) = G'(s)$ and $B(s) \leqslant B'(s)$.*

Moreover, if $\mu^{\mathsf{S}} \neq \mu'^{\mathsf{S}}$ then $(G, B) \neq (G', B')$.

Proof. Consider the game graph $\Gamma_{\mu'^{\mathsf{S}}\chi'^{\mathsf{S}}}$. For every edge $e = (s, s')$, either i) $G(s) > G(s')$, or ii) $G(s) = G(s')$ and $B(s) \geqslant \mathsf{Val}(\partial_e(G(s))) + B(s')$.

Using the same argument as in Lem. 6, we show that $G \geqslant G'$ for all cycles in $\Gamma_{\mu'^{\mathsf{S}}\chi'^{\mathsf{S}}}$, and that $G > G'$ for cycles that did not exist in $\Gamma_{\mu^{\mathsf{S}}\chi^{\mathsf{S}}}$. This proves (1).

Let s be a vertex such that $G(s) = G'(s)$, and let S be a sun in $\Gamma_{\mu'^{\mathsf{S}}\chi'^{\mathsf{S}}}$ such that, $s \in S$. If $s_0, \ldots, s_k$ is the path from s to $\min(S)$ then, for every (s_i, s_{i+1}), $B(s_i) \geqslant \mathsf{Val}(\partial_{(s_i,s_{i+1})}(G(s))) + B(s_{i+1})$. If we sum up, the k inequalities, we get that $B(s)$ is no less then the weight of $s_0, \ldots, s_k$, assuming $\mathsf{Val}(\partial_{(s_i,s_{i+1})}(G(s)))$ to be the weight of edge (s_i, s_{i+1}), which in turn is equal to $G'(s)$. □

Corollary 13. *Solutions to optimality equations for average-price-per-reward games exist.*

Proof. The set of edge strategies for both players is finite. This, together with Thm. 12, guarantees the existence of mutual best response edge strategies. The rest follows from Thm. 11. □

Theorem 14. *Finite definable average-price-per-reward games are decidable.*

Proof. $\mathrm{Opt}(\Gamma)$ is finite hence (G, B) such that $(G, B) \models \mathrm{Opt}(\Gamma)$, is definable (by Rem. 3). □

4 Games on Hybrid Automata with Strong Resets

We introduce hybrid automata with strong resets and define price-reward hybrid games on these automata. The main result is that the hybrid average-price-per-reward games are determined and decidable (Thm. 16). To obtain the result, we reduce hybrid average-price-per-reward games to finite average-price-per-reward games.

Our definition of a hybrid automaton varies from that used in [12,13], as we hide the dynamics of the system into guard functions. This approach allows for cleaner and more succinct notation and exposition, without loss of generality [6].

Price-Reward Hybrid Automata with Strong Resets. Let L be a finite set of *locations*. Fix $n \in \mathbb{N}$ and define the set of *states* $\mathsf{S} = \mathsf{L} \times \mathbb{R}^n$. Let A be a finite set of *actions*, and define the set of *times* $\mathsf{T} = \mathbb{R}_{\oplus}$. We refer to action-time pairs $(a, t) \in \mathsf{A} \times \mathsf{T}$ as *timed actions*. A *price-reward hybrid automaton with strong resets (PRHASR)* $\mathcal{H} = \langle \mathsf{L}, \mathsf{A}, \mathsf{G}, \mathsf{R}, \pi, \kappa \rangle$ consists of finite sets L of *locations* and A of *actions*, a *guard* function $\mathsf{G} : \mathsf{A} \to 2^{\mathsf{S} \times \mathsf{T}}$, a *reset* function $\mathsf{R} : \mathsf{A} \to 2^{\mathsf{S}}$, a continuous *price* function $\pi : \mathsf{S} \times (\mathsf{A} \times \mathsf{T}) \to \mathbb{R}$, and a continuous *reward* function $\kappa : \mathsf{S} \times (\mathsf{A} \times \mathsf{T}) \to \mathbb{R}_{\oplus}$. We say that $\mathcal{H}$ is a *definable PRHASR* if the functions $\mathsf{G}, \mathsf{R}, \pi$, and κ are definable.

For states $s, s' \in \mathsf{S}$ and a timed action $(a, t) \in \mathsf{A} \times \mathsf{T}$, we write $s \xrightarrow{a}_t s'$ iff $(s, t) \in \mathsf{G}(a)$ and $s' \in \mathsf{R}(a)$. If $s, s' \in \mathsf{S}$, $\tau = (a, t) \in \mathsf{A} \times \mathsf{T}$, and $s \xrightarrow{a}_t s'$ then we write $s \xrightarrow{\tau} s'$. We define the *move* function $\mathsf{M} : \mathsf{S} \to 2^{\mathsf{A} \times \mathsf{T}}$ by $\mathsf{M}(s) = \{(a, t) : (s, t) \in \mathsf{G}(a)\}$. Note that M is definable if G is definable. A *run* from state $s \in \mathsf{S}$ is a sequence $\langle s_0, \tau_1, s_1, \tau_2, s_2, \ldots \rangle \in \mathsf{S} \times ((\mathsf{A} \times \mathsf{T}) \times \mathsf{S})^{\omega}$ such that $s_0 = s$, and for all $i \geq 0$, we have $s_i \xrightarrow{\tau_{i+1}} s_{i+1}$.

Hybrid Games with Strong Resets. A *hybrid game with strong resets (HGSR)* $\Gamma = \langle \mathcal{H}, \mathsf{M}^{\mathrm{Min}}, \mathsf{M}^{\mathrm{Max}} \rangle$ consists of a PRHASR $\mathcal{H} = \langle \mathsf{L}, \mathsf{A}, \mathsf{G}, \mathsf{R}, \pi, \kappa \rangle$, a *Min-move* function $\mathsf{M}^{\mathrm{Min}} : \mathsf{S} \to 2^{\mathsf{A} \times \mathsf{T}}$ and a *Max-move* function $\mathsf{M}^{\mathrm{Max}} : \mathsf{S} \times (\mathsf{A} \times \mathsf{T}) \to 2^{\mathsf{A} \times \mathsf{T}}$. We require that for all $s \in \mathsf{S}$, we have $\mathsf{M}^{\mathrm{Min}}(s) \subseteq \mathsf{M}(s)$, and that for all $\tau \in \mathsf{M}^{\mathrm{Min}}(s)$, we have $\mathsf{M}^{\mathrm{Max}}(s, \tau) \subseteq \mathsf{M}(s)$. W.l.o.g., we assume that for all $s \in \mathsf{S}$, we have $\mathsf{M}^{\mathrm{Min}}(s) \neq \emptyset$, and that for all $\tau \in \mathsf{M}^{\mathrm{Min}}(s)$, we have $\mathsf{M}^{\mathrm{Max}}(s, \tau) \neq \emptyset$. If $\mathcal{H}$ and the move functions are definable, then we say that Γ is *definable*.

In the reminder of the paper, we consider price-reward HGSRs. For simplicity, we refer to them as hybrid price-reward games or, when the price-reward aspect is irrelevant, just hybrid games.

A hybrid game is played in rounds. In every round, the following three steps are performed by the two players Min and Max from the current state $s \in \mathsf{S}$.

1. Player Min proposes a timed action $\tau \in \mathsf{M}^{\mathrm{Min}}(s)$.
2. Player Max responds by choosing a timed action $\tau' = (a', t') \in \mathsf{M}^{\mathrm{Max}}(s, \tau)$. This choice determines the price and reward contribution of the round ($\pi(s, \tau')$ and $\kappa(s, \tau')$ respectively).
3. Player Max chooses a state $s' \in \mathsf{R}(a')$, i.e., such that $s \xrightarrow{\tau'} s'$. The state s' becomes the current state for the next round.

A *play* of the game Γ from state s is a sequence $\langle s_0, \tau_1, \tau_1', s_1, \tau_2, \tau_2', s_2, \ldots\rangle \in \mathsf{S} \times ((\mathsf{A} \times \mathsf{T}) \times (\mathsf{A} \times \mathsf{T}) \times \mathsf{S})^\omega$, such that $s_0 = s$, and for all $i \geq 0$, we have $\tau_{i+1} \in \mathsf{M}^{\mathrm{Min}}(s_i)$ and $\tau'_{i+1} \in \mathsf{M}^{\mathrm{Max}}(s_i, \tau_{i+1})$. Note that if $\langle s_0, \tau_1, \tau_1', s_1, \tau_2, \tau_2', s_2, \ldots\rangle$ is a play then the sequence $\langle s_0, \tau_1', s_1, \tau_2', s_2, \ldots\rangle$ is a run of the hybrid automaton $\mathcal{H}$.

A hybrid game with strong resets can be viewed as a game on an infinite price-reward game graph, with fixed costs and rewards assigned to edges. The set of states S' is a subset of: $\mathsf{S} \cup (\mathsf{S} \times (\mathsf{A} \times \mathsf{T})) \cup ((\mathsf{A} \times \mathsf{T}))$. The E' relation is defined as follows: $(s, (s, \tau)) \in \mathsf{E}'$ iff $\tau \in \mathsf{M}^{\mathrm{Min}}(s)$, and $((s, \tau), \tau') \in \mathsf{E}$ iff $\tau' \in \mathsf{M}^{\mathrm{Max}}(s, \tau)$, and $((a', t'), s') \in \mathsf{E}'$ iff $s' \in \mathsf{R}(a')$.

We define $\Gamma' = \langle \mathsf{S}, \mathsf{S}' \setminus \mathsf{S}, \mathsf{E}', \pi', \kappa'\rangle$, where for an edge $e = ((s, \tau), (a', t'))$, we set $\pi'(e) = \pi(s, t')$ and $\kappa'(e) = \kappa(s, t')$, and for all other edges we set them to 0. Additionally, we require that $\mathsf{S}' \setminus \mathsf{S}$ contains all states reachable from S and does not contain those that are not. In the defintion of Γ', we omitted the inputs, as neither the prices nor the rewards depend on them.

Remark 15. For all $(a, t), (a', t') \in \mathsf{S}'$, if $a = a'$ then $(a, t)E' = (a', t')E'$. This is a consequence of the strong reset property of $\mathcal{H}$.

It is clear that plays of Γ directly correspond to runs on Γ'. Moreover, any run of Γ' uniquely determines a run of $\mathcal{H}$. We will use this fact to, lift the concepts introduced for price-reward games to hybrid price-reward games. We will say that the hybrid game Γ has a property P if Γ' has this property.

Hybrid Average-Price-per-Reward Games. In the following, we lift the concept of average-price-per-reward games, as defined in Sec. 2.3, to hybrid price-reward games. We state and prove the main result of the paper:

Theorem 16. *Average-price-per-reward hybrid games are positionally determined and decidable.*

We prove the theorem through a reduction to finite average-price-per-reward games. To obtain the corresponding finite price-reward graph we use an equivalence relation on the state space of the hybrid automaton.

We define the lower and upper payoffs as follows. For a run $\rho = \langle s_0, \tau_1 s_1, \tau_2 \ldots\rangle$ of $\mathcal{H}$, we define the lower payoff P_* and the upper payoff P^* by

$$\mathsf{P}_*(\rho) = \liminf_{n\to\infty} \frac{\sum_{i=0}^{n-1} \pi(s_i, \tau_{i+1})}{\sum_{i=0}^{n-1} \kappa(s_i, \tau_{i+1})} \qquad \mathsf{P}^*(\rho) = \limsup_{n\to\infty} \frac{\sum_{i=0}^{n-1} \pi(s_i, \tau_{i+1})}{\sum_{i=0}^{n-1} \kappa(s_i, \tau_{i+1})}$$

Note that these payoffs are exactly the same, as the average-price-per-reward payoffs for runs starting in $\mathsf{S} \subseteq \mathsf{S}'$ in Γ' (we therefore require that Γ is $\Omega(n)$-divergent and price convergent). This enables us to use the optimality equation characterisation and results from Sec. 2.3. Using Rem. 15 and the fact that A is a finite set, we guarantee that gain has a finite range, and that bias is bounded.

We will also say that $\mathrm{Opt}(\Gamma')$ is the set of optimality equations for the hybrid game Γ, denoted by $\mathrm{Opt}(\Gamma)$. Let $G, B : \mathsf{S} \cup \big(\mathsf{S} \times (\mathsf{A} \times \mathsf{T})\big) \cup \mathsf{A} \to \mathbb{R}$. The optimality equations for Γ' take the following form: if $s \in \mathsf{S}$, then

$$G(s) = \min_{\tau \in \mathsf{M}^{\mathrm{Min}}(s)} \{G(s,\tau)\}, \tag{3}$$

$$B(s) = \inf_{\tau \in \mathsf{M}^{\mathrm{Min}}(s)} \{B(s,\tau) \,:\, G(s,\tau) = G(s)\}; \tag{4}$$

if $s \in \mathsf{S}$ and $\tau \in \mathsf{M}^{\mathrm{Min}}(s)$, then

$$G(s,\tau) = \max_{(a',t') \in \mathsf{M}^{\mathrm{Max}}(s,\tau)} \{G(a')\}, \tag{5}$$

$$B(s,\tau) = \sup_{(a',t') \in \mathsf{M}^{\mathrm{Max}}(s,\tau)} \{\pi(s,a',t') - \kappa(s,a',t') \cdot G(a') + B(a') \,: \, G(a') = G(s,\tau)\}; \tag{6}$$

and if $a \in \mathsf{A}$

$$G(a) = \max_{s \in \mathsf{R}(a)} \{G(s)\}, \qquad B(a) = \sup_{s \in \mathsf{R}(a)} \{B(s) \,:\, G(s) = G(a)\}.$$

The last pair of equations is a generic pair of equations for all states $(a,t) \in \mathsf{S}'$. This is valid by Rem. 15. We have written the equations taking into account the fixed price and rewards in Γ'.

Solving Hybrid Average-Price-per-Reward Games. We show that hybrid average-price-per-reward games are determined and decidable.

In order to establish our results, we use an equivalence relation over the state space of the hybrid game Γ, as introduced in [6]. This relation is of finite index, and its equivalence classes are used to construct a finite price-reward game graph $\widehat{\Gamma}$.

We characterise the game values using optimality equations from Sec. 2.3, and prove that solutions to $\mathrm{Opt}(\widehat{\Gamma})$ coincide with the solutions to $\mathrm{Opt}(\Gamma)$. This, together with the results from Sec. 3 proves that hybrid average-price-per-reward games are determined.

Recall the definition of equivalence relation $\sim$, and the details of the finite graph construction from [6]. We obtain the finite price-reward game graph $\widehat{\Gamma} = (\widehat{\mathsf{S}}^{\mathrm{Min}}, \widehat{\mathsf{S}}^{\mathrm{Max}}, \widehat{\mathsf{E}}, \widehat{\mathcal{I}}, \widehat{\Theta}^{\mathrm{Min}}, \widehat{\Theta}^{\mathrm{Max}}, \widehat{\pi}, \widehat{\kappa})$ from $\Gamma = (\mathcal{H}, \mathsf{M}^{\mathrm{Min}}, \mathsf{M}^{\mathrm{Max}})$ the following way. The finite graph $(\widehat{\mathsf{S}}, \widehat{\mathsf{E}})$ is given by:

$$\widehat{\mathsf{S}} = \mathsf{A} \cup \mathsf{S}/{\sim} \cup \{(Q,a,A') \,:\, Q \in \mathsf{S}/{\sim} \text{ and } (a,A') \in \mathsf{A}^{\mathrm{MinMax}}(Q,\mathsf{T})\},$$

$$\widehat{\mathsf{E}} = \{(a,Q) \,:\, Q \subseteq \mathsf{R}(a)\} \cup \{\big(Q,(Q,a,A')\big) \,:\, (a,A') \in \mathsf{A}^{\mathrm{MinMax}}(Q,\mathsf{T})\} \cup \{\big((Q,a,A'),a'\big) \,:\, a' \in A'\},$$

and the partition of $\widehat{S}$ is given by $\widehat{\mathsf{S}}^{\mathrm{Min}} = \mathsf{S}/{\sim}$ and $\widehat{\mathsf{S}}^{\mathrm{Max}} = \widehat{\mathsf{S}} \setminus \widehat{\mathsf{S}}^{\mathrm{Min}}$. The set of inputs is $\widehat{\mathcal{I}} = \{\mathfrak{z}\} \cup [\mathsf{S} \to \mathsf{A} \times \mathsf{T}] \cup \mathsf{S} \times [\mathsf{A} \times \mathsf{T} \to \mathsf{A} \times \mathsf{T}]$ ($\mathfrak{z}$ serves as a special input for edges that will bear a fixed 0 price and reward). For an edge $e = (Q, a, A')$, let $\widehat{\Theta}^{\mathrm{Min}}(e) \subseteq [Q \to \mathsf{A} \times \mathsf{T}]$ be such that for every $s \in Q$ and $f \in \widehat{\Theta}^{\mathrm{Min}}(e)$, we have that $f(s) \in \mathsf{M}^{\mathrm{Min}}(s)$, and let $\widehat{\Theta}^{\mathrm{Max}}(e) \subseteq Q \times [\mathsf{A} \times \mathsf{T} \to \mathsf{A} \times \mathsf{T}]$ be such that for every $s \in Q$, $\tau \in \mathsf{M}^{\mathrm{Min}}(s)$ and $(s, f) \in \widehat{\Theta}^{\mathrm{Max}}(e)$, we have that $f(\tau) \in \mathsf{M}^{\mathrm{Max}}(s, \tau)$. Let $f \in \widehat{\Theta}^{\mathrm{Min}}(e)$ and $(s, f') \in \widehat{\Theta}^{\mathrm{Max}}(e)$, we define the price (reward) of that edge as $\widehat{\pi}(e)(f, (s, f')) = \pi(s, f'(f(s)))$ ($\widehat{\kappa}(e)(f, (s, f')) = \kappa(s, f'(f(s)))$). For the remaining edges we set $\widehat{\Theta}^{\mathrm{Min}}$ and $\widehat{\Theta}^{\mathrm{Max}}$ to $\{\mathfrak{z}\}$, and their price (reward) to 0.

Theorem 17. *Let Γ be a hybrid average-price-per-reward game and let $(\widehat{G}, \widehat{B}) \models Opt(\widehat{\Gamma})$. If $G, B : \mathsf{S} \cup \big(\mathsf{S} \times (\mathsf{A} \times \mathsf{T})\big) \cup \mathsf{A} \to \mathbb{R}$ are such that $G(a) = \widehat{G}(a)$ and $B(a) = \widehat{B}(a)$ for all $a \in \mathsf{A}$, and satisfy equations (3–6), then $(G, B) \models Opt(\Gamma)$.*

Corollary 18. *Definable average-price-per-reward hybrid games with strong resets are decidable.*

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
2. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
3. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE Computer Society Press, Los Alamitos (1996)
4. Lafferriere, G., Pappas, G.J., Sastry, S.: O-minimal hybrid systems. Mathematics of Control, Signals, and Systems 13(1), 1–21 (2000)
5. Gentilini, R.: Reachability problems on extended O-minimal hybrid automata. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 162–176. Springer, Heidelberg (2005)
6. Bouyer, P., Brihaye, T., Jurdziński, M., Lazić, R., Rutkowski, M.: Average-price and reachability-price games on hybrid automata with strong resets. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 63–77. Springer, Heidelberg (2008)
7. Brihaye, T., Michaux, C.: On the expressiveness and decidability of o-minimal hybrid systems. Journal of Complexity 21(4), 447–478 (2005)
8. Bouyer, P., Brinksma, E., Larsen, K.G.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design 32(1), 2–23 (2008)
9. Abdeddam, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theoretical Computer Science, 272–300 (2006)
10. Church, A.: Logic, arithmetic and automata. In: Proceedings of the International Congress of Mathematicians, pp. 23–35 (1962)
11. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
12. Bouyer, P., Brihaye, T., Chevalier, F.: Control in o-minimal hybrid systems. In: LICS 2006. LNCS, pp. 367–378. Springer, Heidelberg (2006)

13. Bouyer, P., Brihaye, T., Chevalier, F.: Weighted o-minimal hybrid systems are more decidable than weighted timed automata! In: Artemov, S.N., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 69–83. Springer, Heidelberg (2007)
14. Jurdziński, M., Trivedi, A.: Reachability-time games on timed automata. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 838–849. Springer, Heidelberg (2007)
15. Brihaye, T., Michaux, C., Rivière, C., Troestler, C.: On o-minimal hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 219–233. Springer, Heidelberg (2004)
16. Brihaye, T.: A note on the undecidability of the reachability problem for o-minimal dynamical systems. Mathematical Logic Quarterly 52, 165–170 (2006)
17. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press (1951)
18. Weihrauch, K.: Computable Analysis. Springer, Heidelberg (2000)
19. Meer, K., Michaux, C.: A survey on real structural complexity theory. Bulletin of the Belgian Mathematical Society 4, 113–148 (1997); Journées Montoises (Mons, 1994)
20. Blum, L., Shub, M., Smale, S.: On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. American Mathematical Society. Bulletin. New Series 21, 1–46 (1989)

Abstraction Refinement for Probabilistic Software

Mark Kattenbelt, Marta Kwiatkowska, Gethin Norman, and David Parker

Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD

Abstract. We present a methodology and implementation for verifying ANSI-C programs that exhibit probabilistic behaviour, such as failures or randomisation. We use abstraction-refinement techniques that represent probabilistic programs as Markov decision processes and their abstractions as stochastic two-player games. Our techniques target quantitative properties of software such as "the maximum probability of file-transfer failure" or "the minimum expected number of loop iterations" and the abstractions we construct yield lower and upper bounds on these properties, which then guide the refinement process. We build upon state-of-the-art techniques and tools, using SAT-based predicate abstraction, symbolic implementations of probabilistic model checking and components from GOTO-CC, SATABS and PRISM. Experimental results show that our approach performs very well in practice, successfully verifying actual networking software whose complexity is significantly beyond the scope of existing probabilistic verification tools.

1 Introduction

Software model checking techniques have become increasingly sophisticated in recent years. Witness for example the success of the SLAM project, used to identify bugs in Windows device drivers. This technology is based on *predicate abstraction* [1] and *counterexample-guided abstraction-refinement* (CEGAR) [2], which are used to construct increasingly precise finite-state abstractions of programs to either demonstrate the violation of a safety property (e.g. a buffer overflow) or guarantee the absence of such faults.

In this paper, we present novel techniques for verification of software that exhibits *probabilistic* behaviour, for example due to interaction with components prone to failures or due to the use of randomisation. We target ANSI-C programs, extending the language with two probabilistic functions: `coin`(p), which returns 1 with probability p and 0 with probability $1-p$; and `prob`(n), which returns an integer between 0 and $n-1$ uniformly at random. These provide a natural way of modelling both failures (e.g. a function call to open a network connection which fails with probability p) and randomisation (e.g. selecting a random pivot to sort a list of n items). We also provide functions to model nondeterministic behaviour, such as calls to underspecified procedures or program input.

Our approach is based on *probabilistic model checking*, a generalisation of model checking for systems that exhibit stochastic behaviour. It is applied to

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 182–197, 2009.

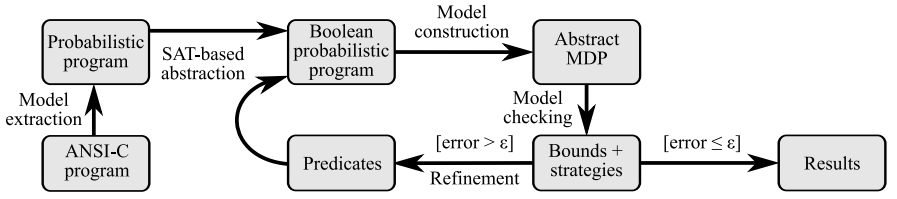


Fig. 1. Abstraction-refinement loop for probabilistic programs

state transition systems augmented with probabilistic information, such as Markov decision processes (MDPs). The properties to be verified are *quantitative* in nature and, since MDPs model both probability and nondeterminism, relate to best- or worst-case bounds on behaviour, e.g. "the maximum probability of file-transfer failure" or "the minimum expected number of loop iterations". Various tools are available for probabilistic model checking, such as PRISM, MRMC and LiQuor, and the techniques have been successfully applied to a wide range of applications from security to biological modelling. They have yet, however, to be used in the context of real programming languages.

We present a quantitative analogue of the well-known CEGAR loop (see Figure 1), which successively refines an abstraction of a concrete model until it is sufficiently precise. The combination of probabilistic and nondeterministic behaviour is naturally modelled with MDPs. The underlying theory is based on representing abstractions of MDPs as two-player stochastic games [3], which use the two players to distinguish the nondeterminism of the concrete system and that introduced during abstraction. We use SAT-based predicate abstraction [4] to construct, from a concrete probabilistic program, an abstraction in the form of a *Boolean probabilistic program*, whose semantics is the stochastic game abstraction. Model checking the game [3] yields lower and upper bounds on a quantitative property of the original MDP (such as "the minimum probability that the program terminates successfully" or "the maximum expected number of function calls during program execution") and strategies (for the two players) that achieve the bounds. Although the analysis does not yield counterexamples (in the sense of a trace to an error state), the bounds and strategies provide both a quantitative measure of the precision of the abstraction and, when necessary, provide a means of refining the abstraction.

Related work. The closest work is [5], which proposes a CEGAR framework for predicate abstraction of MDPs described in a simple guarded-command language. This verifies or refutes properties of the form "the maximum probability of error is at most p" for a probability threshold p. In [5], abstractions are also MDPs (as proposed in [6]), and only upper bounds on maximum probabilities are computed. So, to refute a property, probabilistic counterexamples [7] (comprising multiple paths whose combined probability exceeds p) are generated. If these paths are spurious, they are used to generate further predicates using interpolation. Perhaps the most important distinguishing feature of our work is the use of two-player games as abstractions. These provide *lower and upper bounds*,

which avoids enumerating a set of paths, which can be large or even infinite (resulting in non-termination). The bounds also enable a *quantitative* approach: we target properties without thresholds such as "what is the maximum probability of error?". A second important difference is that we use real programming languages, rather than guarded-commands. Lastly, we also consider rewards.

Other abstraction-refinement techniques for probabilistic systems have been proposed. In [6], abstractions of MDPs are refined by state-space partitioning but, like [5], this yields only one-sided bounds. Magnifying lens abstraction [8] partitions an MDP and analyses each one separately but, since it relies on building the full concrete model, even a symbolic implementation [9] is unlikely to scale to real software. In [10], a counterexample-based abstraction-refinement technique for planning problems is proposed; however, there is no implementation to test this on practical examples.

In [11], a framework is described for analysing probabilistic programs based on expectation transformers, but this is not applied to real software and does not include an automated step for refining abstractions. Another approach to abstracting probabilistic models is to label transitions with intervals, e.g. [12], but this has only been applied to models without nondeterminism. In earlier work [13], we applied the game-based abstraction of [3] to predicate abstraction, but not for source code and without refinement.

Another important direction for the verification of probabilistic software is the extension of abstract interpretation to the probabilistic setting [14,15,16], although these approaches have yet to be combined with refinement. Other approaches to the probabilistic verification of imperative languages include APEX [17], which performs equivalence checking for a simple procedural language, and the tool LiQuor [18], whose modelling language Probmela includes imperative language style constructs. Neither approach uses abstraction.

Contributions. The precise contributions of this paper are the following:

- we present the first abstraction-refinement techniques for probabilistic systems that are specifically targeted at real programming languages;
- we describe a complete implementation of these techniques, built using state-of-the-art tools and techniques, and demonstrate its applicability on several real software case studies that cannot be verified with existing tools;
- we improve upon existing approaches by using game-based abstraction to obtain both lower and upper bounds on quantitative properties.

2 Background

A probability distribution over a set S is a function $\lambda : S \to [0,1]$ satisfying $\sum_{s \in S} \lambda(s){=}1$. Let $\mathsf{dist}(S)$ denote the set of all distributions over S. We use the notation $p_1 : s_1 + \cdots + p_n : s_n$ for the distribution $\lambda \in \mathsf{dist}(S)$ such that $\lambda(s_i) = p_i$. For a set X, $x \in X$, $\lambda \in \mathsf{dist}(S)$ and $\Lambda \in \mathcal{P}(\mathsf{dist}(S))$, let $x \boxtimes \lambda \in \mathsf{dist}(X \times S)$ denote the distribution where $(x \boxtimes \lambda)(x, s){=}\lambda(s)$ and $x \boxtimes \Lambda$ denote the set $\{x \boxtimes \lambda \mid \lambda \in \Lambda\}$.

Definition 1 (Markov decision process). *A* Markov decision process *(MDP) is a tuple* $\mathsf{M} = \langle S, S_i, \delta \rangle$*, where* S *is a set of* states*,* $S_i \subseteq S$ *are* initial states *and* $\delta : S \to \mathcal{P}(\mathsf{dist}(S))$ *is a* probabilistic transition function*, which maps each state to a finite, non-empty set of probability distributions over states.*

An MDP's behaviour is both probabilistic and nondeterministic. A transition $s \to\!\!\!\!\!\!\lambda\; s'$ from state s is made by first nondeterministically selecting a distribution $\lambda \in \delta(s)$, and then selecting a successor state s' with probability $\lambda(s')$. A *path* is a sequence of transitions. A state is *reachable* if there is a path to it from an initial state. Under an *adversary*, which resolves all nondeterminism, we can define a probability measure over paths [19]. For a target set $\mathcal{F} \subseteq S$, we then define the *minimum* and *maximum probability*, under any adversary, of reaching $\mathcal{F}$ from state s, denoted $\mathbf{p}_s^-(\mathcal{F})$ and $\mathbf{p}_s^+(\mathcal{F})$ respectively. By also associating rewards (non-negative real values) with transitions, we can also define the *minimum* and *maximum expected reward* of reaching $\mathcal{F}$.

Definition 2 (Abstract MDP). *An* abstract MDP *is a tuple* $\hat{\mathsf{M}} = \langle \hat{S}, \hat{S}_i, \hat{\delta} \rangle$*, where* $\hat{S}$ *is a set of* abstract states*,* $\hat{S}_i \subseteq \hat{S}$ *are* initial abstract states *and* $\hat{\delta} : \hat{S} \to \mathcal{P}(\mathcal{P}(\mathsf{dist}(\hat{S})))$ *is an* abstract probabilistic transition function*, which maps each state to a set of sets of distributions over states.*

The underlying semantics of an abstract MDP is a two-player stochastic game [20]. A transition $\hat{s} \to\!\!\!\!\!\!\langle \Lambda, \lambda \rangle\; \hat{s}'$ includes two successive nondeterministic choices: first, a set of distributions $\Lambda \in \hat{\delta}(\hat{s})$ is chosen by player 1; then, an element $\lambda \in \Lambda$ is selected by player 2. The successor $\hat{s}'$ is chosen with probability $\lambda(\hat{s}')$. Similarly to MDPs, under *strategies* for players 1 and 2, which resolve all nondeterminism, we can define a probability measure over paths. For target $\hat{\mathcal{F}} \subseteq \hat{S}$, we define both the probability and expected reward of reaching $\hat{\mathcal{F}}$ from a state $\hat{s}$ or choice Λ when both players minimise, player 1 minimises and 2 maximises, player 1 maximises and 2 minimises and both maximise. For reachability probabilities, we denote these $\mathbf{p}_{\hat{s}}^{--}(\hat{\mathcal{F}})$, $\mathbf{p}_{\hat{s}}^{-+}(\hat{\mathcal{F}})$, $\mathbf{p}_{\hat{s}}^{+-}(\hat{\mathcal{F}})$ and $\mathbf{p}_{\hat{s}}^{++}(\hat{\mathcal{F}})$ respectively.

As proposed in [3], abstract MDPs are used to represent abstractions of MDPs. The key idea is to separate the two forms of nondeterminism: the first choice in a transition (player 1) represents nondeterminism caused by abstraction; the second choice (player 2) corresponds to the nondeterminism of the original MDP. For an MDP M, the construction of its abstraction is based on an *abstraction function* $\alpha : S \to \hat{S}$ from concrete to abstract states. We lift α to distributions and sets and in the obvious way, e.g. $\alpha(\lambda)(\hat{s}) = \sum_{\alpha(s)=\hat{s}} \lambda(s)$.

Definition 3 (Abstraction of MDPs [3]). *Given an MDP* $\mathsf{M} = \langle S, S_i, \delta \rangle$ *and abstraction function* $\alpha : S \to \hat{S}$*, the* abstraction of M under α *is the abstract MDP* $\alpha(\mathsf{M}) = \langle \hat{S}, \alpha(S_i), \hat{\delta} \rangle$ *where for any* $\hat{s} \in \hat{S}$ *we have that* $\Lambda \in \hat{\delta}(\hat{s})$ *if and only if there exists* $s \in S$ *such that* $\alpha(s){=}\hat{s}$ *and* $\Lambda{=}\alpha(\delta(s))$*.*

The abstraction $\alpha(\mathsf{M})$ of M yields lower and upper bounds on probabilities and expected rewards of M [3]. For example, for any $s \in S$ and $\mathcal{F} \subseteq S$:

$$\mathbf{p}^{--}_{\alpha(s)}(\alpha(\mathcal{F})) \leqslant \mathbf{p}^{-}_{s}(\mathcal{F}) \leqslant \mathbf{p}^{+-}_{\alpha(s)}(\alpha(\mathcal{F}))$$
$$\mathbf{p}^{-+}_{\alpha(s)}(\alpha(\mathcal{F})) \leqslant \mathbf{p}^{+}_{s}(\mathcal{F}) \leqslant \mathbf{p}^{++}_{\alpha(s)}(\alpha(\mathcal{F}))$$

Algorithms for computing these measures for MDPs and abstract MDPs can be found in e.g. [21] and [22], respectively.

3 Probabilistic Programs

In this section, we define *probabilistic programs*. Since these are both probabilistic and nondeterministic in nature, their semantics are given in terms of MDPs.

Let $\mathcal{U}$ denote a *data universe*, that is, the set of all possible *data valuations*. Given an expression $\mathtt{E}$ over $\mathcal{U}$ and a valuation $u \in \mathcal{U}$, $\mathtt{E}(u)$ denotes the evaluation of $\mathtt{E}$ on u. For an l-value $\mathtt{x}$ and an expression $\mathtt{E}$, $u[\mathtt{x}\mapsto\mathtt{E}]$ denotes the valuation derived from u by setting $\mathtt{x}$ to $\mathtt{E}(u)$ and $\mathrm{Type}(\mathtt{x})$ the set of all values of the same type as $\mathtt{x}$. The set of *commands* $\mathcal{C}_{\mathcal{U}}$ over $\mathcal{U}$ consists of: conditional statements $[\mathtt{B}]$, deterministic assignments $\mathtt{x}{=}\mathtt{E}$, probabilistic assignments $\mathtt{i}{=}\mathtt{coin}(p)$ and $\mathtt{i}{=}\mathtt{prob}(n)$, nondeterministic assignments $\mathtt{i}{=}\mathtt{ndet}(n)$ and $\mathtt{i}{=}\mathtt{ndet}()$, where $\mathtt{B}$ is a Boolean expression over $\mathcal{U}$, $\mathtt{E}$ is an expression over $\mathcal{U}$, $\mathtt{x}$ is an l-value, $\mathtt{i}$ is an integer l-value, $p \in (0,1)$ and $n \in \mathbb{N}$. We use GOTO-CC [23] to transform programs such that all expressions are side-effect free and all assignments are type-consistent. The l-values in deterministic assignments can be of any valid type, including pointers, structures and arrays.

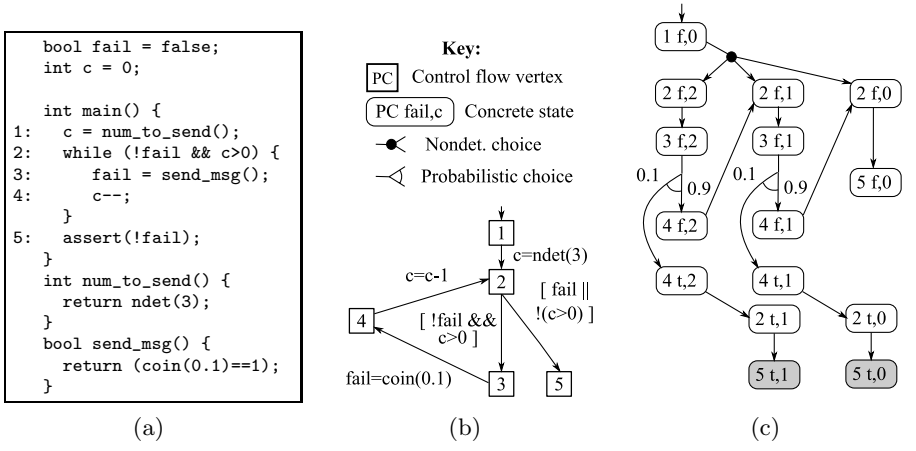
Definition 4 (Probabilistic program). *A* probabilistic program P *is a tuple* $\langle \mathcal{U}, \langle V, E\rangle, v_i, \mathcal{L}\rangle$ *where* $\mathcal{U}$ *is a data universe,* $\langle V, E\rangle$ *is a finite directed (control-flow) graph with initial vertex* v_i *and* $\mathcal{L} : E \to \mathcal{C}_{\mathcal{U}}$ *labels edges with commands.*

We assume that if an outgoing edge from a vertex v is labelled with a conditional, then so are all other outgoing edges from v and, for each $u \in \mathcal{U}$, precisely one of these conditions holds. Any other vertex has only a single outgoing edge. Therefore, each vertex is associated with a single type of command.

During program extraction function calls are inlined; thus we do not support unbounded recursion. We also do not consider dynamic memory allocation or floating point arithmetic. We assume a conventional model checker guarantees the absence of any undefined behaviour, e.g. a null-pointer dereference, during the evaluation of expressions and l-values. We deal with pointers through static points-to analysis augmented with (dynamic) information using predicates [4]. The semantics of non-probabilistic commands are captured with transitions that occur with probability 1.

Definition 5 (Probabilistic program semantics). *Let* $\mathsf{P}{=}\langle \mathcal{U}, \langle V, E\rangle, v_i, \mathcal{L}\rangle$ *be a probabilistic program. The semantics of* P *is the MDP* $[\![\mathsf{P}]\!]{=}\langle V{\times}\mathcal{U}, \{v_i\}{\times}\mathcal{U}, \delta\rangle$ *where for any* $\langle v, u\rangle \in V{\times}\mathcal{U}$*:*

$$\delta(v,u) = \left\{ v' \boxtimes \lambda \;\middle|\; \langle v, v'\rangle \in E,\ \lambda \in [\![\mathcal{L}\langle v, v'\rangle]\!](u) \right\}$$



(a) (b) (c)

Fig. 2. Simple example: (a) C code; (b) probabilistic program; (c) MDP semantics.

and $[\![\texttt{cmd}]\!] : \mathcal{U} \to \mathcal{P}(\mathsf{dist}(\mathcal{U}))$ *is the semantics of command* `cmd` *such that for* $u \in \mathcal{U}$*:*

$$
\begin{aligned}
[\![\texttt{[B]}]\!](u) &= \{1 : u\} \textit{ if } \texttt{B}(u) \textit{ and } \emptyset \textit{ otherwise} \\
[\![\texttt{x=E}]\!](u) &= \{1 : u[\,\texttt{x} \mapsto \texttt{E}(u)\,]\} \\
[\![\texttt{i=coin}(p)]\!](u) &= \{(1-p) : u[\texttt{i} \mapsto \texttt{0}] + p : u[\texttt{i} \mapsto \texttt{1}]\} \\
[\![\texttt{i=prob}(n)]\!](u) &= \{\tfrac{1}{n} : u[\texttt{i} \mapsto \texttt{0}] + \cdots + \tfrac{1}{n} : u[\texttt{i} \mapsto n\texttt{-1}]\} \\
[\![\texttt{i=ndet}(n)]\!](u) &= \{1 : u[\texttt{i} \mapsto \texttt{0}], \ldots, 1 : u[\texttt{i} \mapsto n\texttt{-1}]\} \\
[\![\texttt{i=ndet()}]\!](u) &= \{1 : u[\texttt{x} \mapsto \texttt{val}] \mid \texttt{val} \in \mathrm{Type}(\texttt{x})\}\,.
\end{aligned}
$$

Example 1. Figure 2 shows a fragment of C code, corresponding probabilistic program and MDP semantics. The code comprises a loop which tries to send `c` messages, `c` being obtained by calling `num_to_send()`, which nondeterministically returns 0, 1 or 2. A message is sent by calling `send_msg()`, which fails with probability 0.1. Once a transmission fails, the loop terminates. The maximum probability of any transmission failing (i.e. of reaching control-flow location 5 with `fail` equal to true) is 0.19 and occurs when `c` is set to 2.

4 Abstraction of Probabilistic Programs

In practice, constructing the concrete semantics of all but the simplest programs is intractable. Hence, in order to verify real programs, it becomes essential to consider *abstraction*. We adopt the approach of [24] and use *Boolean probabilistic programs*, which retain the same control-flow structure as their concrete counterpart but abstract the concrete data universe $\mathcal{U}$ to a finite Boolean abstraction induced by set of *predicates* (Boolean expressions) over $\mathcal{U}$ [1].

Definition 6 (Boolean probabilistic program). *A* Boolean probabilistic program B *is a tuple* $\langle \Phi, \langle V, E\rangle, v_i, \mathcal{T}\rangle$ *where* Φ *is a set of* n *(quantifier-free)*

predicates, $\langle V, E\rangle$ *is a directed (control-flow) graph with initial vertex* v_i *and* $\mathcal{T} : E \to (\mathbb{B}^n \to \mathcal{P}(\mathcal{P}(\mathsf{dist}(\mathbb{B}^n))))$ *is an* abstract probabilistic transition function.

The semantics of a concrete probabilistic program is an MDP; the semantics of its abstraction, a Boolean probabilistic program, is an abstract MDP. Conventional (non-probabilistic) Boolean programs are typically used to represent existential abstractions [25] where both concrete and abstract semantic models are labelled transition systems. In this case, a Boolean program can be seen as a special instance of a (concrete) program. In our setting this does not hold since the semantic models of concrete and abstract programs differ. Hence, we define Boolean probabilistic programs directly in terms of a mapping $\mathcal{T}$ rather than through commands (as we did with $\mathcal{L}$ for probabilistic programs).

Definition 7 (Boolean probabilistic program semantics). *The semantics of a Boolean probabilistic program* $\mathsf{B}{=}\langle \Phi, \langle V, E\rangle, v_i, \mathcal{T}\rangle$ *is the abstract MDP* $[\![\mathsf{B}]\!]{=}\langle V{\times}\mathbb{B}^n, \{v_i\}{\times}\mathbb{B}^n, \hat{\delta}\rangle$ *where* $n = |\Phi|$ *and for any* $\langle v, a\rangle \in V{\times}\mathbb{B}^n$:

$$\hat{\delta}(v, a) = \left\{ v' \boxtimes \Lambda \;\middle|\; \langle v, v'\rangle \in E,\ \Lambda \in \mathcal{T}(v, v')(a) \right\}.$$

Given a probabilistic program $\mathsf{P}{=}\langle \mathcal{U}, \langle V, E\rangle, v_{\mathrm{i}}, \mathcal{L}\rangle$ and predicates $\Phi{=}\{\phi_1, \ldots, \phi_n\}$ over the data universe $\mathcal{U}$, we now show how to construct the corresponding abstract Boolean probabilistic program. The abstraction function $\alpha : \mathcal{U} \to \mathbb{B}^n$ is given by $\alpha(u){=}\langle \phi_1(u), \ldots, \phi_n(u)\rangle$; we lift α to distributions and sets and to $V{\times}\mathcal{U}$ by letting $\alpha\langle v, u\rangle{=}\langle v, \alpha(u)\rangle$. For any $a = \langle b_1, \ldots, b_n\rangle \in \mathbb{B}^n$, let $a[i] = b_i$.

Definition 8 (Abstraction of probabilistic programs). *Given a probabilistic program* $\mathsf{P}{=}\langle \mathcal{U}, \langle V, E\rangle, v_i, \mathcal{L}\rangle$ *and set of predicates* Φ *with abstraction function* α*, the* abstraction of P under Φ *is given by the Boolean probabilistic program* $\alpha(\mathsf{P}){=}\langle \Phi, \langle V, E\rangle, v_i, \mathcal{T}\rangle$ *where for any* $e \in E$ *and* $a \in \mathbb{B}^n$*:* $\Lambda \in \mathcal{T}(e)(a)$ *if and only if there exists* $u \in \mathcal{U}$ *such that* $\alpha(u){=}a$ *and* $\Lambda{=}\alpha([\![\mathcal{L}(e)]\!](u)){\neq}\emptyset$*.*

Applying MDP abstraction (Definition 3) to the semantics of a concrete program (Definition 5) yields the same abstraction as the Boolean program (Definitions 8 and 7). This is because, although Definition 8 applies the abstraction per control-flow edge, there is no nondeterminism between edges in the concrete program.

Example 2. Figure 3(a) shows a representation of the Boolean probabilistic program obtained by abstracting the program from Example 1 using predicates `fail` and (`c==0`). We use $\phi{=}{*_1}$ and $\phi{=}{*_2}$ to describe the abstract probabilistic transition function in which the value of the predicate ϕ is determined by player 1 or player 2, respectively. For example, if $a{=}\langle b_1, b_2\rangle \in \mathbb{B}^2$ and $\lambda_{\mathrm{f}}, \lambda_{\mathrm{t}}$ be the distributions $1{:}(b_1, \mathrm{f})$ and $1{:}(b_1, \mathrm{t})$, then (`c==0`)$=*_1$ on edge e indicates that $\mathcal{T}(e)(a) = \{\{\lambda_{\mathrm{f}}\}, \{\lambda_{\mathrm{t}}\}\}$, whereas (`c==0`)$=*_2$ means that $\mathcal{T}(e)(a) = \{\{\lambda_{\mathrm{f}}, \lambda_{\mathrm{t}}\}\}$. Figure 3(b) shows the abstract MDP semantics of the Boolean probabilistic program. Each abstract state is labelled with lower/upper bounds on the maximum probability of reaching control-flow location 5 with variable `fail` equal to true.

SAT-based abstraction. In order to construct the abstraction of a probabilistic program, we adopt the SAT-based techniques of [4], in which the basic idea is

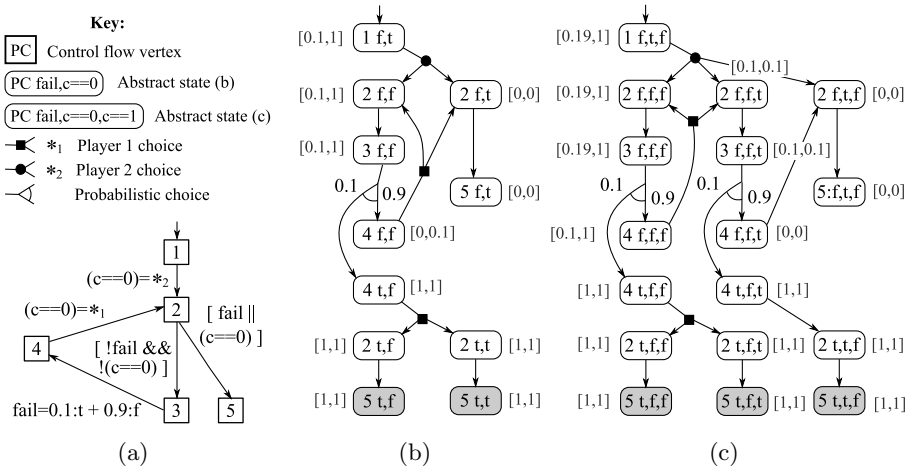


Fig. 3. Abstractions for Example 1: (a) & (b) Boolean probabilistic program and abstract MDP for initial abstraction; (c) abstract MDP for refined abstraction

to construct the abstract transition relation for each edge of a program's control-flow graph by formulating it as a Boolean satisfiability problem. Each satisfiable instance corresponds to an element of the transition relation; all such instances can be enumerated efficiently by a SAT-solver. An important advantage of this approach is that it allows a detailed bit-level semantics of the source code.

Our setting is slightly different: our abstractions are Boolean probabilistic programs and so we construct abstract probabilistic transition functions, rather than abstract transition relations. Despite this, fundamental similarities remain: the use of Boolean programs means that the abstraction for each command can be built in isolation; and the definition of abstraction can be phrased as an existential satisfiability problem.

Consider a probabilistic program $\mathsf{P}{=}\langle \mathcal{U}, \langle V, E\rangle, v_{\mathrm{i}}, \mathcal{L}\rangle$ and set of n predicates Φ. To construct the abstraction we need only construct the abstract probabilistic transition function $\mathcal{T}(e)$ for each edge $e \in E$. Recall from Definition 8 that, for each $a \in \mathbb{B}^n$, $\mathcal{T}(e)(a)$ returns a set of probability distributions over $\mathbb{B}^n$ where $\Lambda \in \mathcal{T}(e)(a)$ if and only if there exists $u \in \mathcal{U}$ such that $\alpha(u){=}a$ and $\Lambda{=}\alpha([\![\mathcal{L}(e)]\!](u)){\neq}\emptyset$. We now formulate $\mathcal{T}(e)$ as a satisfiability problem whose structure is dependent on the command labelling e.

Conditionals. If e is labelled $[\mathtt{B}]$, then $[\![\mathcal{L}(e)]\!](u)$ equals $\{1{:}u\}$ if $\mathtt{B}(u)$ and $\emptyset$ otherwise, and hence $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{1 : a\}$ and:

$$\exists\, u \in \mathcal{U} \,.\, \big(\, \alpha(u){=}a \wedge \mathtt{B}(u) \,\big)$$

Deterministic Assignments. If e is labelled $\mathtt{x}{=}\mathtt{E}$, then $[\![\mathcal{L}(e)]\!](u){=}\{1{:}u[\mathtt{x}{\mapsto}\mathtt{E}(u)]\}$, and hence $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{1 : a'\}$ for some $a' \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U} \,.\, \big(\, \alpha(u){=}a \wedge \alpha(u[\,\mathtt{x}{\mapsto}\mathtt{E}(u)\,]){=}a' \,\big)$$

Probabilistic assignments. If e is labelled $\mathtt{i}{=}\mathtt{coin}(p)$, then $[\![\mathcal{L}(e)]\!](u){=}\{(1{-}p) : u[\mathtt{i}{\mapsto}\mathtt{0}] + p : u[\mathtt{i}{\mapsto}\mathtt{1}]\}$, and $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{(1{-}p){:}a_0 + p{:}a_1\}$ for some $a_0, a_1 \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U} \,.\, \big(\, \alpha(u){=}a \wedge \alpha(u[\mathtt{i}{\mapsto}\mathtt{0}]){=}a_0 \wedge \alpha(u[\mathtt{i}{\mapsto}\mathtt{1}]){=}a_1 \,\big)$$

The case when e is labelled $\mathtt{i}{=}\mathtt{prob}(n)$ follows similarly.

Nondeterministic assignments. If e is labelled $\mathtt{i}{=}\mathtt{ndet}(n)$, then $[\![\mathcal{L}(e)]\!](u) = \{1{:}u[\mathtt{i}{\mapsto}\mathtt{0}], \ldots, 1{:}u[\mathtt{i}{\mapsto}n{-}\mathtt{1}]\}$, and therefore $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{1 : a_0, \ldots, 1 : a_{n-1}\}$ for some $a_0, \ldots, a_{n-1} \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U} \,.\, \big(\, \alpha(u){=}a \;\wedge\; \alpha(u[\mathtt{i}{\mapsto}\mathtt{0}]){=}a_0 \;\wedge \ldots \wedge\; \alpha(u[\mathtt{i}{\mapsto}\mathtt{n}{-}\mathtt{1}]){=}a_{n-1} \,\big)$$

Computing $\mathcal{T}(e)$ in this way for $\mathtt{i}{=}\mathtt{ndet}(n)$ with large n can result in intractable SAT formulas. The same is true if we adopt a similar approach for assignments $\mathtt{x}{=}\mathtt{ndet}()$. So, for such commands, we make the assumption that $\mathcal{T}(e)(a)$ contains a single set, Λ say. Since $[\![\mathcal{L}(e)]\!](u) = \{1 : u[\mathtt{x}{\mapsto}\mathtt{val}] \mid \mathtt{val} \in \mathrm{Type}(\mathtt{x})\}$, we have $\lambda \in \Lambda$ if and only if $\lambda{=}1 : a'$ for some $a' \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U} \,.\, \exists\, \mathtt{val} \in \mathrm{Type}(\mathtt{x}) \,.\, \big(\, \alpha(u){=}a \wedge \alpha(u[\,\mathtt{x}{\mapsto}\mathtt{val}\,]){=}a' \,\big)$$

The above assumption holds if Φ can be partitioned into $\{\Phi_{\mathtt{x}}, \Phi{\setminus}\Phi_{\mathtt{x}}\}$, where predicates in $\Phi_{\mathtt{x}}$ refer only to $\mathtt{x}$, and those in $\Phi{\setminus}\Phi_{\mathtt{x}}$ are not influenced by $\mathtt{x}$. Fortunately, this case turns out to be sufficient in practice. Assignments of the form $\mathtt{x}{=}\mathtt{ndet}()$ typically model operating system calls that nondeterministically succeed or fail and the actual values being assigned are irrelevant to the property under consideration. The same assumption is used for $\mathtt{i}{=}\mathtt{ndet}(n)$ with large n.

5 Abstraction Refinement

In order to make our abstraction techniques practically applicable, it is essential to develop *refinement* techniques that can automatically construct an abstraction which is sufficiently precise for verification but also small enough for efficient analysis. In conventional CEGAR approaches, this is based on the generation of counterexamples (paths to an error state), which are either *feasible* (i.e. a concretisation exists and the safety property is refuted) or *spurious* (in which case, the counterexample is used to generate additional predicates for refinement).

The crucial difference in our setting is that model checking is quantitative: our aim is not just to establish the absence/existence of a path to a target, but rather to compute quantitative properties, e.g. the minimum probability or maximum reward of reaching a target. The abstraction-refinement loop we propose (as illustrated earlier in Figure 1) is based on iterative refinement of an abstraction (an abstract MDP) until the lower and upper bounds for the property of interest differ by less than some threshold ε. In this section, we describe how the abstract MDP yields predicates that can be used to refine the abstraction.

Predicate discovery. We describe the case for maximum probabilities (the process for minimum probabilities and expected rewards is identical). Therefore suppose we have a probabilistic program $\mathsf{P} = \langle \mathcal{U}, \langle V, E\rangle, v_\mathrm{i}, \mathcal{L}\rangle$, target $\mathcal{F} \subseteq V \times \mathcal{U}$ and predicates Φ with abstraction function α such that $\alpha(\mathsf{P})$ is not sufficiently precise for some initial state, i.e. there exists $a \in \mathbb{B}^n$ such that:

$$\mathbf{p}^{++}_{\langle v_\mathrm{i}, a\rangle}(\alpha(\mathcal{F})) - \mathbf{p}^{-+}_{\langle v_\mathrm{i}, a\rangle}(\alpha(\mathcal{F})) > \varepsilon\,. \tag{1}$$

Recall that model checking the abstraction $\alpha(\mathsf{P})$ also yields strategies that achieve the lower and upper bounds. A strategy tells us how nondeterminism is resolved, i.e. it gives for each abstract state $\langle v, a\rangle$ an element Λ of $\hat{\delta}\langle v, a\rangle$ in the abstract MDP. Each such choice Λ encodes a subset of the concrete states represented by $\langle v, a\rangle$, namely the set $\{\langle v, u\rangle \in V \times \mathcal{U} \mid \alpha(u){=}a,\ \alpha(\delta\langle v, u\rangle){=}\Lambda\}$.

Based on results from [3,26], the inequality in (1) guarantees that there exists an abstract state $\langle s^\star, a^\star\rangle$ and *distinct* choices Λ^- and Λ^+ made by lower and upper bound strategies in $\langle s^\star, a^\star\rangle$ such that either $\mathbf{p}^{-+}_{\Lambda^-}(\alpha(\mathcal{F})) < \mathbf{p}^{-+}_{\Lambda^+}(\alpha(\mathcal{F}))$ or $\mathbf{p}^{++}_{\Lambda^-}(\alpha(\mathcal{F})) < \mathbf{p}^{++}_{\Lambda^+}(\alpha(\mathcal{F}))$. We call $\langle s^\star, a^\star\rangle$ a *refinable state*. Our aim is to eliminate the choice between Λ^- and Λ^+, through a predicate that separates the concrete states corresponding to these two choices. For example, if $\mathbf{p}^{-+}_{\Lambda^-}(\alpha(\mathcal{F})) < \mathbf{p}^{-+}_{\Lambda^+}(\alpha(\mathcal{F}))$, then choosing Λ^+ makes the lower bound higher. Hence we can improve the lower bound of the states encoded by Λ^+ by eliminating the choice.

Below, we describe how to generate a new predicate, based on the command associated with the control location of the refinable state $\langle v^\star, a^\star\rangle$. By construction, $v^\star$ either has one or more outgoing edges labelled with conditionals or a single outgoing edge $\langle v^\star, v\rangle$ labelled with an assignment.

Conditionals. If the outgoing edges of $v^\star$ are conditionals, then $\Lambda^-{=}\{1{:}\langle v^-, a^\star\rangle\}$ and $\Lambda^+{=}\{1{:}\langle v^+, a^\star\rangle\}$ for some distinct v^- and v^+ such that $\langle v^\star, v^-\rangle$ and $\langle v^\star, v^+\rangle$ are labelled with conditionals ($[\mathtt{B}^-]$ and $[\mathtt{B}^+]$ say). We add $\mathtt{B}^-$ to Φ. By assumption on probabilistic programs, at most one of $\mathtt{B}^-$ and $\mathtt{B}^+$ is satisfiable in the concretisations of an abstract state, eliminating the choice between Λ^- and Λ^+.

Deterministic Assignments. If $\langle v^\star, v\rangle$ is labelled $\mathtt{x{=}E}$, then $\Lambda^-{=}\{1{:}\langle v, a^-\rangle\}$ and $\Lambda^+{=}\{1{:}\langle v, a^+\rangle\}$ for some $a^-, a^+ \in \mathbb{B}^n$. Since Λ^- and Λ^+ are distinct there exists a predicate $\phi_i \in \Phi$ such that $a^-[i]{\neq}a^+[i]$. We add the predicate $\mathrm{WP}(\phi_i, \mathtt{x{=}E})$. By definition, this predicate is satisfied if, after executing the assignment $\mathtt{x{=}E}$, ϕ_i holds, i.e. $\mathrm{WP}(\phi_i, \mathtt{x{=}E})(u)$ if and only if $\phi_i(u[\mathtt{x}{\mapsto}\mathtt{E}])$. If $\langle v^\star, u^-\rangle$ is encoded by Λ^-, then $\alpha(u^-[\mathtt{x}{\mapsto}\mathtt{E}]){=}\Lambda^-$, and hence $\phi_i(u^-[\mathtt{x}{\mapsto}\mathtt{E}])$ if and only if $a^-[i]$. A similar argument holds for states encoded by Λ^+. As $a^-[i]{\neq}a^+[i]$, the new predicate is either satisfied by all states encoded by Λ^- and none by Λ^+ or vice versa, and therefore $\mathrm{WP}(\phi_i, \mathtt{x{=}E})$ eliminates the choice between Λ^- and Λ^+.

Probabilistic assignments. If $\langle v^\star, v\rangle$ is labelled $\mathtt{i{=}coin}(p)$, then, Λ^- and Λ^+ are of the form $\{(1{-}p){:}\langle v, a_0^-\rangle + p{:}\langle v, a_1^-\rangle\}$ and $\{(1{-}p){:}\langle v, a_0^+\rangle + p{:}\langle v, a_1^+\rangle\}$. Since $\Lambda^-{\neq}\Lambda^+$, there exists $\phi_i \in \Phi$ such that $a_j^-[i]{\neq}a_j^+[i]$ for some $0{\leqslant}j{\leqslant}1$ and we add $\mathrm{WP}(\phi_i, \mathtt{x{=}j})$ to Φ which, by similar arguments to above, removes the choice between Λ^- and Λ^+. The case when $\langle v^\star, v\rangle$ is labelled $\mathtt{i{=}prob}(n)$ follows similarly.

Nondeterministic assignments. By construction an assignment `x=ndet()` consists of a single choice, hence $\langle v^\star, v\rangle$ cannot be labelled `x=ndet()`. If $\langle v^\star, v\rangle$ is labelled `i=ndet(n)`, then Λ^- and Λ^+ are of the form $\{1{:}\langle v, a_0^-\rangle, \ldots, 1{:}\langle v, a_{n-1}^-\rangle\}$ and $\{1{:}\langle v, a_0^+\rangle, \ldots, 1{:}\langle v, a_{n-1}^+\rangle\}$ respectively. Since $\Lambda^- \neq \Lambda^+$, there exists $\phi_i \in \Phi$ such that $a_j^-[i] \neq a_j^+[i]$ for some $0 \leqslant j \leqslant n-1$ and we add the predicate $\mathrm{WP}(\phi_i, \texttt{x=j})$.

As in conventional CEGAR, our method is incomplete due to the use of WP-based abstraction refinement [27]. However, as we will show later, our approach successfully finds suitable abstractions in practice.

Example 3. Consider the program of Example 1 and the abstraction from Example 2 (Figures 3(a) and 3(b)). The abstract state (4 f,f) is the only one with both a player 1 choice and differing bounds (0 if branching right to (2 f,t) and 0.1 if branching left to (2 f,f)), i.e. it is the only possible refinable state. The command for control-flow vertex 4 is the deterministic assignment `c=c−1` and the predicate `(c==0)` differs between (2 f,t) and (2 f,f) so our new predicate is $\mathrm{WP}(\texttt{(c==0)}, \texttt{c=c−1})$, i.e. `(c==1)`. The abstraction under the predicates $\texttt{fail}, \texttt{(c==0)}, \texttt{(c==1)}$ is shown in Figure 3(c). We see that the bounds on the maximum probability for the initial state have tightened from $[0.1, 1]$ to $[0.19, 1]$. A further refinement (on the same control-flow vertex) would result in an abstraction equivalent to the original MDP, yielding exact bounds $[0.19, 0.19]$.

Extensions. We investigate several extensions to the refinement loop.

Refinable state selection. Our refinement scheme can be applied to any refinable state $\langle v^\star, a^\star\rangle$. Hence, we consider two heuristics for choosing a refinable state: "maximum error" (pick a state with the greatest difference in lower and upper bounds, aiming to refine the abstraction where it is least precise); "nearest" (pick a state closest to the initial states). In addition, since model checking an abstract MDP (which determines the refinable states) is relatively expensive, we consider refining *multiple states* within a single iteration of the refinement loop.

Avoiding unreachable states. Although a refinable state $\langle v^\star, a^\star\rangle$ is always reachable in the abstract MDP, there is no guarantee that any concretisation of $\langle v^\star, a^\star\rangle$ is reachable in the concrete MDP. Hence, refining $\langle v^\star, a^\star\rangle$ could add unnecessary complexity to the abstraction. To avoiding this, we employ *spurious path removal*: we find an abstract path to $\langle v^\star, a^\star\rangle$ and use SAT-based symbolic simulation to check if a concretisation of the path exists. If not, in addition we use conventional weakest precondition-based refinement to eliminate the path.[1]

Predicate initialisation. Conventional (non-probabilistic) CEGAR can be used to check the existence of a path to the target. The predicates generated during this process are likely to form a subset of those found by our refinement approach and can potentially be discovered more efficiently in this fashion. Hence, we consider employing existing efficient CEGAR tools to generate an initial set of predicates.

[1] This approach cannot guarantee to detect if $\langle v^\star, a^\star\rangle$ is unreachable, since doing so amounts to fully verifying a safety property with conventional CEGAR tools.

Predicate localisation. It is well known that successful implementations of predicate abstraction compute abstractions efficiently because they keep the number of relevant predicates small, e.g. by exploiting locality [28]. We apply similar ideas to our approach, only adding discovered predicates to locations where they are required, based on a backwards control-flow traversal from the refinable state. This also allows us to take an incremental approach to building the abstraction, reusing the previous abstraction for locations with no new predicates.

6 Implementation and Results

We have built a complete implementation of the techniques described. Model extraction from C code is done using an extension of GOTO-CC [23]. Predicate abstraction was implemented using components from the SATABS tool [29] and MiniSAT SAT solver. Model checking of stochastic games is done with extensions of the symbolic engines of the probabilistic model checker PRISM [30].

We illustrate the practicality of our approach by studying its performance on several case studies. We consider two networking utilities: an ICMP ping client and a TFTP client.[2] Both are approximately 1KLOC in size and feature complex programming constructs such as arrays, pointers and function pointers. Low-level kernel and networking functions are replaced with stubs whose behaviour is either probabilistic (e.g. opening a socket) or nondeterministic (e.g. user input). We also consider ANSI-C versions of several protocols used as probabilistic model checking benchmarks: Herman's self-stabilisation (from APEX [17]), Zeroconf and the Bounded Retransmission Protocol. All programs are available[3] and the properties verified for each are listed in Figure 4.

We ran experiments for several different configurations of the options described in the previous section ("maximum error" and "nearest" refinable state selection; with and without spurious path removal and predicate initialisation). Table 1 presents detailed statistics for the fastest verification run on each example; Table 2 compares the different configurations. All experiments were run on an Intel Core Duo 2 (T7200) with 2GB RAM. The CEGAR loop terminated when the (relative) error was below $\varepsilon{=}10^{-4}$. All timings are in seconds.

Overall performance. The results demonstrate that our method verifies a wide range of programs and quantitative properties in an efficient and fully automatic manner. This is particularly impressive for the more complex ping and TFTP utilities. The tables show that the number of refinement iterations and predicates are relatively low. The difference between the total and average numbers of predicates indicates that the use of predicate localisation is essential. With regards to timings (Table 1), we see that abstraction and refinement are in most cases efficient, whereas the model checking phase is the most expensive. One reason for this is that the numerical solution process is relatively expensive (compared to the model checking required in conventional, non-probabilistic

[2] Based on based on GNU Inetutils 1.5 and TFTP-HPA 0.48, respectively.

[3] All programs are available at `www.prismmodelchecker.org/files/vmcai09/`.

ping	A	"max. probability of not receiving a reply to an echo request"
	B	"max. prob. of establishing connectivity with packet loss following two requests"
	C	"max. expected number of echo requests required to establish connectivity"
tftp	A	"max. probability of establishing a write request"
	B	"max. probability of successfully transferring some file data"
	C	"max. expected amount of data that is sent before timeout"
her	A	"min. probability of terminating in a stable state"
	B	"max. expected number of rounds before termination"
zcnf	A	"min. probability of configuring with a fresh IP"
	B	"max. expected number of probes"
brp	A	"max. probability of the receiving nothing while a chunk was sent"
	B	"max. probability of the sender reporting uncertainty"

Fig. 4. List of properties verified

Table 1. Experimental results: detailed results for fastest verification run

		refinement iterations	predicates total (avg.)	timing breakdown init.	abstr.	check	refine	total time
ping	A	4	33 (7.82)	56%	15%	27%	2%	**15.5**
	B	31	45 (9.27)	-	48%	45%	7%	**87.2**
	C	12	16 (2.73)	-	17%	68%	15%	**5.37**
tftp	A	11	39 (10.7)	32%	15%	48%	5%	**57.8**
	B	28	51 (12.0)	-	46%	45%	9%	**96.5**
	C	22	35 (9.22)	-	17%	75%	8%	**64.4**
her	A	18	24 (7.08)	-	17%	82%	1%	**33.5**
	B	2	40 (11.1)	7%	2%	91%	<1%	**259**
zcnf	A	9	11 (3.94)	-	9%	85%	6%	**1.97**
	B	9	10 (3.81)	4%	10%	77%	9%	**1.43**
brp	A	5	6 (6.00)	-	50%	31%	19%	**0.71**
	B	7	7 (7.00)	-	44%	44%	12%	**1.34**

CEGAR) and is performed twice (for the lower and upper bound). Also, due to predicate localisation, abstractions can be constructed incrementally.

Extensions. Table 2 shows the performance of the configurable options of our implementation. [4] For *refinable state selection*, although neither policy for choosing a refinable state is consistently better, "nearest" seems the sensible default as there are several cases where it is significantly faster. In particular, the "maximum error" policy for property B of the ping utility takes over an hour due to repeatedly choosing refinable states with no reachable concretisations. This problem is successfully resolved using *spurious path removal*. However we see that, in several cases, this produces a significant slow-down. This is because, although symbolic simulation itself is relatively fast, the predicates it adds result in slower model checking times. The situation is worse for the "maximum error" policy as the generated paths are longer and give more predicates. Employing

[4] Since refining multiple refinable states did not yield significant improvements in performance, we have omitted the results for this extension.

Table 2. Experimental results: comparison of techniques. Timeout '-' is 1 hour.

		default				spurious path removal				predicate initialisation			
		max error		nearest		max error		nearest		max error		nearest	
		pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time
ping	A	33 (6.9)	18.4	33 (6.3)	16.6	34 (6.4)	41.2	33 (6.3)	28.8	33 (7.8)	**15.5**	33 (7.8)	15.5
	B	-	-	45 (9.3)	**87.2**	54 (12)	803	56 (12)	627	54 (13)	305	53 (13)	301
	C	16 (2.7)	**5.37**	20 (3.2)	7.97	17 (3.1)	6.92	22 (3.6)	15.1	19 (3.3)	6.59	19 (3.3)	7.39
tftp	A	41 (11)	95.4	41 (11)	111	52 (11)	262	52 (12)	142	39 (11)	64.3	39 (11)	**57.8**
	B	51 (12)	173	51 (12)	**96.5**	62 (13)	1,680	62 (13)	227	56 (15)	194	56 (15)	161
	C	35 (8.4)	69.0	35 (9.2)	**64.4**	46 (8.9)	888	42 (8.8)	81.7	37 (9.2)	85.1	37 (10)	86.3
her	A	25 (6.9)	49.9	24 (7.1)	**33.5**	32 (7.6)	215	26 (6.6)	66.1	40 (11)	36.6	40 (11)	36.3
	B	34 (8.9)	520	27 (7.8)	751	39 (10)	974	32 (8.4)	619	40 (11)	**259**	40 (11)	263
zcnf	A	11 (3.9)	**1.97**	11 (3.9)	1.98	11 (3.9)	2.07	11 (3.9)	2.12	11 (4.1)	2.10	11 (4.1)	2.17
	B	10 (3.8)	1.44	10 (3.8)	1.45	10 (3.8)	1.51	10 (3.8)	1.54	10 (3.8)	**1.43**	10 (3.8)	1.54
brp	A	7 (7.0)	0.92	6 (6.0)	0.72	8 (8.0)	0.91	6 (6.0)	**0.71**	15 (15)	3.53	15 (15)	3.53
	B	7 (7.0)	**1.34**	10 (10)	2.12	27 (27)	6.98	12 (12)	2.29	-	-	-	-

predicate initialisation (through SATABS), increases efficiency on several examples. Often in cases where it performs best (e.g. property A of the ping and TFTP utilities), there are relatively few paths to the target state, so refining based on a single path is productive. On examples with a large number of such paths, using the game-based refinement alone performs better. In particular, for property B of BRP, predicate initialisation is very slow because the target is only reachable through very long paths but our method only needs to consider a small number of loop iterations for sufficiently tight bounds.

Related tools. Finally, we briefly compare our implementation with some related tools. Although a direct comparison with [5] is not possible (due to the difference in input language), we note that property A of BRP (called p4 in [5]) takes under a second here and about 5 seconds in [5] on a comparable machine. Also, we obtain sufficiently tight bounds discovering 6 predicates, compared to 25 in [5]. In [17], the Herman case study is tested on the APEX tool but the focus is different (contextual equivalence) and no times are given. PRISM [30] can be applied to the last three case studies and is faster, but only by using manually constructed abstractions. The more complex programs, the ping and TFTP utilities, are significantly beyond the scope of PRISM.

7 Conclusions

We have presented a novel abstraction-refinement method for verification of software with probabilistic behaviour. Our approach uses two-player stochastic games, SAT-based predicate abstraction and probabilistic model checking. The use of game-based abstractions allows us to compute lower and upper bounds on quantitative properties, which form the basis of our abstraction-refinement loop. We have demonstrated the applicability of our approach by successfully verifying a selection of case studies, including several complex programs well beyond the reach of state-of-the-art probabilistic model checkers such as PRISM.

We plan to extend this work in several directions. These include investigating the use of imprecise abstractions, an approach frequently take in conventional software verification to improve efficiency, and developing techniques to handle probabilistic choices over larger ranges. We also hope to improve the way in which loops are dealt with. Currently, the abstractions we construct include an explicit representation of the loop, which is required to compute, for example, the probability of the loop terminating. We plan to investigate the use of existing techniques such as ranking functions to improve efficiency in this area.

Acknowledgments. The authors are supported in part by EPSRC grants EP/D07956X and EP/D076625. We would also like to thank Daniel Kroening for his advice and support regarding SATABS.

References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
2. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
3. Kwiatkowska, M., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: Proc. QEST 2006, pp. 157–166. IEEE, Los Alamitos (2006)
4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. FMSD 25(2-3), 105–127 (2004)
5. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
6. D'Argenio, P., Jeannet, B., Jensen, H., Larsen, K.: Reachability analysis of probabilistic systems by successive refinements. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
7. Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)
8. de Alfaro, L., Roy, P.: Magnifying-lens abstraction for Markov decision processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 325–338. Springer, Heidelberg (2007)
9. Roy, P., Parker, D., Norman, G., de Alfaro, L.: Symbolic magnifying lens abstraction in Markov decision processes. In: Proc. QEST 2008. IEEE, Los Alamitos (2008)
10. Chatterjee, K., Henzinger, T., Jhala, R., Majumdar, R.: Counterexample-guided planning. In: Proc. UAI 2005, pp. 104–111 (2005)
11. McIver, A., Morgan, C.: Abstraction, refinement and proof for probabilistic systems. Springer, Heidelberg (2004)
12. Huth, M.: On finite-state approximants for probabilistic computation tree logic. Theoretical Computer Science 346(1), 113–134 (2005)
13. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Game-based probabilistic predicate abstraction in PRISM. In: Proc. QAPL 2008(2008)
14. Pierro, A.D., Wiklicky, H.: Concurrent constraint programming: Towards probabilistic abstract interpretation. In: Proc. PPDP 2000, pp. 127–138. ACM Press, New York (2000)

15. Monniaux, D.: Abstract interpretation of programs as Markov decision processes. Science of Computer Programming 58(1-2), 179–205 (2005)
16. Smith, M.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. In: Proc. QAPL 2008 (2008)
17. Legay, A., Murawski, A., Ouaknine, J., Worrell, J.: On automated verification of probabilistic programs. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008)
18. Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: Proc. QEST 2006, pp. 131–132. IEEE, Los Alamitos (2006)
19. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains, 2nd edn. Springer, Heidelberg (1976)
20. Shapley, L.: Stochastic games. Proc. Nat. Acad. Science 39, 1095–1100 (1953)
21. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis (1997)
22. Condon, A.: On algorithms for simple stochastic games. Advances in computational complexity theory 13, 51–73 (1993)
23. GOTO-CC, `http://www.verify.ethz.ch/goto-cc/`
24. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI 2001, pp. 203–213 (2001)
25. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)
26. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Technical Report RR-08-06, Oxford University Computing Laboratory (2008)
27. Jhala, R., McMillan, K.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
28. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: Proc. POPL 2004, pp. 232–244. ACM Press, New York (2004)
29. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005)
30. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)

Finding Concurrency-Related Bugs Using Random Isolation

Nicholas Kidd[1], Thomas Reps[1,3], Julian Dolby[2], and Mandana Vaziri[2]

[1] University of Wisconsin
{kidd,reps}@cs.wisc.edu
[2] IBM T.J. Watson Research Center
{dolby,mvaziri}@us.ibm.com
[3] GrammaTech, Inc.

Abstract. This paper describes the methods used in Empire, a tool to detect concurrency-related bugs, namely atomic-set serializability violations in Java programs. The correctness criterion is based on *atomic sets* of memory locations, which share a consistency property, and *units of work*, which preserve consistency when executed sequentially. Empire checks that, for each atomic set, its units of work are serializable. This notion subsumes data races (single-location atomic sets), and serializability (all locations in one atomic set).

To obtain a sound, finite model of locking behavior for use in Empire, we devised a new abstraction principle, *random isolation*, which allows *strong updates* to be performed on the abstract counterpart of each randomly-isolated object. This permits Empire to track the status of a Java lock, even for programs that use an unbounded number of locks. The advantage of random isolation is that properties proved about a randomly-isolated object can be generalized to all objects allocated at the same site. We ran Empire on eight programs from the ConTest benchmark suite, for which Empire detected numerous violations.

1 Introduction

This paper describes the methods used in Empire, a tool to detect atomic-set serializability violations in concurrent Java programs. *Atomic-set serializability* [1] is a data-centric correctness criterion for concurrent programs. It is based on the notion of an *atomic set* of memory locations, which specifies the *existence* of an invariant or consistency property. Associated with atomic sets are *units of work*, which preserve atomic-set consistency when executed sequentially. Atomic-set serializability means that, for each atomic set, its units of work are *serializable*, where an execution is serializable if it is equivalent to a serial execution in which each thread's units of work are executed with no interleavings from other threads.

Atomic-set serializability subsumes other correctness criteria for concurrent systems, such as data-race freedom (single-field atomic sets), and serializability (all fields in one atomic set). Such other criteria ignore the intended relationships

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 198–213, 2009.

that may exist between shared memory locations, and thus may not accurately reflect the intentions of the programmer about correct behavior.

EMPIRE is a tool to statically detect atomic-set serializability violations (henceforth referred to as "violations") in concurrent Java programs. A key challenge that we faced was how to create a sound, finite model of a Java program's locking behavior that is capable of tracking the status of a Java lock, for programs that use an unbounded number of locks. To address this issue, we devised a new abstraction principle, *random isolation*, which has two key advantages:

1. It allows *strong updates* to be performed on the abstract counterparts of each randomly-isolated object, which permits EMPIRE to track the status of the Java lock associated with a randomly-isolated object.
2. It allows properties proved about a randomly-isolated object to be generalized to *all* objects allocated at the same site.

EMPIRE is based on the result that executions that are not atomic-set serializable can be characterized by a set of problematic interleaving scenarios [1]: an execution that is free of all of these scenarios is guaranteed to be atomic-set serializable.[1] In EMPIRE, a problematic interleaving scenario with respect to a set of shared memory locations is used as an input specification to a model checker. Specifically, EMPIRE translates a concurrent Java program into a communicating pushdown system (CPDS) [2,3], and translates the scenario into a *violation monitor* that checks for the occurrence of the scenario, and runs concurrently with the other CPDS processes. Once the translation is performed, the generated CPDS is fed into a CPDS model checker [3].

Previous work [1] addressed the inference of synchronization and appropriate placement of locks, given annotations for atomic sets and units of work. A second paper [4] focused on legacy code and checking whether an existing multi-threaded program is appropriately synchronized, by dynamically detecting the occurrence of problematic interleaving scenarios. The work on EMPIRE complements these other approaches by providing a method to statically check Java programs for problematic interleaving scenarios. EMPIRE's checking algorithm uses the CPDS model checker's semi-decision procedure to (symbolically) consider multiple executions of the program. This is in contrast with the dynamic-detection approach [4], which only looks at one execution at a time.

EMPIRE has two modes of operation. For code that satisfies certain properties,[1] it can verify the absence of violations. If the properties are not met, then it can miss errors, and thus operates as a bug detector, rather than a verification tool. The contributions of our work can be summarized as follows:

– We introduce a new abstraction principle, *random isolation*, which allows *strong updates* to be performed on the abstract counterparts of each *randomly-isolated object*. With this approach, properties proved about a randomly-isolated object can be generalized to all objects allocated at the

[1] This result relies on an assumption that programs do not always satisfy: a unit of work that writes to one location of an atomic set, writes to all locations in that atomic set.

same site. Random-isolation is a generic abstraction that should be applicable in many other contexts, such as typestate verification [5] and other temporal-safety analyses for object-oriented programs.
- We present a static technique for detecting atomic-set serializability violations in concurrent Java programs. The method uses random isolation to obtain a sound, finite model of locking behavior that, in many circumstances, is able to track the status of a Java lock precisely, even for programs that use an unbounded number of locks.
- We implemented these techniques in EMPIRE, and ran EMPIRE on eight programs from the ConTest benchmark suite[6], for which EMPIRE detected numerous violations, including ones involving multiple locations.

2 Overview

Fig. 1 is a simple Java program inspired by one of the ConTest programs [6]. There are two classes, `Shop` and `Client`. The intention of the programmer is that the method `Client.buy()` executes atomically, so that when `getItem()` is called on the parameter `Shop s`, `s` is non-empty. However, this intention is not implemented correctly: method `buy()` is synchronized on `this` and not on `s`, hence multiple clients of the same shop could interleave. Fig. 1 shows an interleaved program execution illustrating this concurency-related bug. After thread 2 finishes the call to `getItem()`, the field `items` is -1, which leads thread 1 to access the array `storage` with a negative index. This problem can be fixed by taking a lock on `s` in the body of `buy()`. Notice that there is no data race in this program, so traditional race detectors would not catch this bug.

This concurrency-related bug is an instance of an atomic-set-serializability violation. In this code, fields `items` and `storage` form an *atomic set*: they are meant to be updated atomically due to a consistency property. Each method of class `Shop` is a *unit of work* for this atomic set: when executed sequentially, it preserves the consistency property. In addition, the `buy()` method of `Client` must manipulate the parameter `Shop s` atomically. It is therefore a unit of work for the atomic set of `s`. The interleaved execution of Fig. 1 shows that the two units of work representing the method `buy()` are not serializable: i.e., the execution may produce a final state different from that of any serial execution of the two methods.

Atomic-set serializability is characterized by a set of problematic interleaving scenarios: i.e., an execution that does not contain any of the scenarios is atomic-set-serializable. In the example, the interleaved execution contains the following problematic scenario: $R_1(l_1), W_2(l_2), W_2(l_1), R_1(l_2)$, where l_1 (l_2) is bound to i (s). (See [1] for a complete list of these scenarios.) Notice that atomic-set-serializability is finer-grained than most notions of serializability because it is per atomic set, rather than embracing the whole heap.

EMPIRE detects atomic-set-serializability violations by statically checking for problematic interleaving scenarios. The user provides a concurrent Java program Prog, and specifies an allocation site ψ for a class T in Prog. (This is exemplified

```
class Shop {
  Object[] storage = new Object[10];
  int items = -1;
  public static Shop makeShop(){
    return new_ψ Shop(); // ⟵ ψ
  }
  public synchronized Object getItem(){
    Object res = storage[items];
    storage[items--] = null;
    return res;
  }
  public synchronized void put(Object o){
    storage[++items] = o;
  }
  public synchronized boolean empty(){
    return (items == -1);
  }
}
```

```
class Client {
  public synchronized boolean buy(Shop s){
    if(!s.empty()) { s.getItem(); return true; }
    else return false;
  }
  public static Client makeClient(){
    return new Client();
  }
  public static void main(String[] args){
    Shop shop1 = Shop.makeShop();
    Shop shop2 = Shop.makeShop();
    Client client1 = makeClient();
    Client client2 = makeClient();
    new Thread("1") { client1.buy(shop1); }
    new Thread("2") { client2.buy(shop1); }
  }
}
```

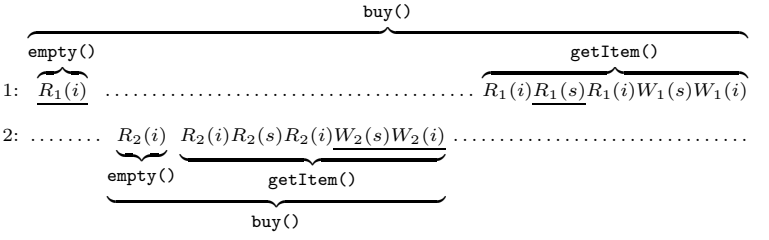


Fig. 1. Example Program. R and W denote a read and write access, respectively. i and s denote fields `item` and `storage`, respectively. Subscripts are thread ids.

by the $\texttt{new}_\psi$ statement in Fig. 1 for the class `Shop`.) EMPIRE uses the default assumptions of [4]: T has one atomic set containing all of T's declared fields (one atomic set per object), and every `public` method of T is a unit of work for that atomic set. Additionally, any method that takes a T object as a parameter is also a unit of work. EMPIRE then performs violation detection, focusing on the atomic sets of objects that can be allocated at ψ.

EMPIRE performs violation detection in four stages. First, a source-to-source transformation is applied to the (potentially) infinite-data program Prog to prepare it for abstraction, obtaining a program Prog^* (§3). Second, a finite-data abstraction is created for translating Prog^* into EMPIRE's intermediate modeling language EML (§4). Third, from this EML program, EMPIRE generates CPDSs to model the program and monitor for problematic interleaving scenarios (§5). Fourth, state-space exploration is carried out on the generated CPDSs.

The challenge is to design a finite-data abstraction such that (i) the set of behaviors of the abstracted program is a sound overapproximation of the set of behaviors of the original Java program, and (ii) the abstraction is able to disallow certain thread interleavings by modeling the program's synchronization.

A natural choice for a finite-data abstraction is the *allocation-site abstraction* [7]. Given an allocation site ψ for class T, let $\mathsf{Conc}(\psi)$ denote the set of

all concrete objects of type T that can be allocated at ψ. The allocation-site abstraction uses a single abstract object $\varsigma^\sharp_\psi$ to summarize all of the concrete objects in $\mathsf{Conc}(\psi)$. Thus, for each field f defined by T, field $\varsigma^\sharp_\psi.f$ is a summary field for the set of fields $\{\varsigma.f \mid \varsigma \in \mathsf{Conc}(\psi)\}$. Because the program has a finite number of program points, and each class defines a finite number of fields, this results in a finite-data abstraction.

For such an approach to be sound, an analysis generally has to perform *weak updates* on each summary object. That is, information for the summary object must be *accumulated* rather than overwritten. A *strong update* of the abstract state generally can only be performed when the analysis can prove that there is exactly one object allocated at ψ, i.e., $|\mathsf{Conc}(\psi)| = 1$.

Violation detection is concerned with tracking reads and writes to the fields of the T objects allocated at ψ. The allocation-site abstraction is a sound overapproximation for modeling reads and writes because a read (write) to the abstract field $\varsigma^\sharp_\psi.f$ corresponds to a possible read (write) to $\varsigma.f$, for all $\varsigma \in \mathsf{Conc}(\psi)$.

Violation detection must also model program synchronization. EMPIRE accomplishes this by defining locks in the EML program that correspond to the objects of Prog^*. There are two possibilities for defining the semantics of an EML lock. The first is to interpret a lock acquire as a *strong update*, i.e., the program has definitely acquired a particular lock. This would correspond to acquiring the locks of *all possible* instances in $\mathsf{Conc}(\psi)$, which in most circumstances would be unsound. In the example of Fig. 1, this interpretation of locking combined with the allocation-site abstraction would preclude the interleaved program execution that contains the bug, because the two `Client` objects would effectively get the same lock, and the two `buy()` methods would execute without interleaving. The second possibility for defining the semantics of EML locks is to interpret lock acquire as a *weak update*, i.e., the program may have acquired a particular lock. This semantics is sound, but the analysis gains no precision on locking behavior, since all lock operations are possible rather than definite. In general, this possibility would greatly increase the number of false positives. For instance, in the example of Fig. 1, if we were to fix the code by adding an additional synchronization block on `s` inside the body of `buy()`, analysis would still report a bug because locking behavior was modeled imprecisely.

Our solution is to use a new abstraction: *random-isolation abstraction*, which is a novel extension of allocation-site abstraction. The extension involves randomly isolating one of the concrete objects allocated at allocation site ψ and tracking it specially in the abstraction. Whereas allocation-site abstraction would associate one summary object to ψ, random isolation associates two objects to ψ: one summary and one non-summary. Because one is a non-summary object, it is safe to perform strong updates to its (abstract) state. The EML model will have an EML lock for each non-summary object, on which strong updates—definite lock acquires and releases—are performed. In constrast, because sound tracking of the lock state for a summary object generally would result in $\top$, our models have no locks on summary objects: their modeled behaviors are not restricted

by synchronization primitives. This provides a sound, finite model of the locking behavior of Prog^*. (It is an over-approximation because the absence of locks on summary objects causes them to gain *additional* behaviors.)

The essence of random isolation can be captured via a simple source-to-source transformation. Consider the following code fragment.

$$\mathsf{public\ static\ Shop\ makeShop()\ \{\ return\ new}_{\psi}\ \mathsf{Shop();\ \}} \tag{1}$$

Random isolation involves transforming the allocation statement into

$$(\mathsf{rand()\ \&\&\ test\text{-}and\text{-}set}(G_\psi))\ ?\ \mathsf{new}_{\psi_\star}\ \mathsf{Shop()}\ :\ \mathsf{new}_{\psi}\ \mathsf{Shop();}. \tag{2}$$

The site ψ from code fragment (1) is transformed into a conditional-allocation site, where the conditional "tests-and-sets" a newly introduced global flag G_ψ. The global flag G_ψ ensures that only one object can ever be allocated at the generated site $\psi_\star$. This has two benefits: (i) because abstract object $\varsigma^\sharp_{\psi_\star}$ is a non-summary object, strong updates can be performed on it, and (ii) because concrete object $\varsigma_{\psi_\star}$ is chosen randomly, every property proven to hold for $\varsigma^\sharp_{\psi_\star}$ must also hold for *every* concrete object $\varsigma_\psi \in \mathsf{Conc}(\psi)$.

3 Random-Isolation Abstraction

The *random-isolation abstraction* is motivated by the following observation:

Observation 1. *The concrete objects that can be allocated at a given allocation site* ψ, $\mathsf{Conc}(\psi)$, *cannot be distinguished by the allocation-site abstraction.*

Obs. 1 says that if one chooses to isolate a *random concrete* object ς from the summary object $\varsigma^\sharp_\psi$, the allocation-site abstraction would not be able to distinguish the randomly-chosen concrete object from any of the other concrete objects that are summarized by $\varsigma^\sharp_\psi$.

Random isolation extends allocation-site abstraction in two ways. First, whereas allocation-site abstraction uses one abstract object $\varsigma^\sharp_\psi$ to summarize the concrete objects $\mathsf{Conc}(\psi)$, random-isolation abstraction associates *two* abstract objects with ψ: $\varsigma^\sharp_\psi$ and $\varsigma^\sharp_{\psi_\star}$. Second, the global boolean flag G_ψ records whether the *randomly-isolated object* has been allocated or not. This eliminates the possibility that the concretization of the special abstract object $\varsigma^\sharp_{\psi_\star}$ is the empty set, and enforces isolation, which gives us *Random-Isolation Principle 1*:

Random-Isolation Principle 1 (Updates). *Let* $\varsigma_\star \in \mathsf{Conc}(\psi)$ *be a randomly-isolated concrete object. Because* $\varsigma_\star$ *is modeled by a special abstract object* $\varsigma^\sharp_{\psi_\star}$, *the random-isolation abstraction enables an analysis to perform strong updates on the state of* $\varsigma^\sharp_{\psi_\star}$.

Random isolation also provides a powerful methodology for proving properties of a program: a proof that a property ϕ holds for $\varsigma^\sharp_{\psi_\star}$ proves that ϕ holds for all $\varsigma \in \mathsf{Conc}(\psi)$. Consider a concrete trace of the program in which a concrete object

ς' is allocated at a dynamic instance of ψ, and ϕ does not hold for ς'. Because of random isolation, the randomly-isolated object $\varsigma_{\psi_\star}$ is just as likely to be ς' as it is to be any other concrete object. Thus, the prover must consider the case that $\varsigma_{\psi_\star}$ is ς'. Because the property holds for $\varsigma^\sharp_{\psi_\star}$, and because $\varsigma^\sharp_{\psi_\star}$ represents ς' in the trace under consideration, then the property must also hold for ς', which is a contradiction. This gives us *Random-Isolation Principle 2*:

Random-Isolation Principle 2 (Proofs). *Given a property ϕ and site ψ, a proof that ϕ holds for the randomly-isolated abstract object $\varsigma^\sharp_{\psi_\star}$ proves that ϕ holds for every object that is allocated at ψ. That is, $\phi(\varsigma^\sharp_{\psi_\star}) \rightarrow (\forall_{\varsigma \in \mathsf{Conc}(\psi)}.\phi(\varsigma))$.*

Before describing the technical details of how we implemented random isolation, we highlight the benefits of random isolation for performing violation detection. Because of random isolation, the state of the Java lock that is associated with the random instance $\varsigma_{\psi_\star}$ can be modeled precisely by the state of the special abstract object $\varsigma^\sharp_{\psi_\star}$. That is, the acquiring and releasing of the lock for $\varsigma_{\psi_\star}$ by a thread of execution can be modeled by a strong update on the state of $\varsigma^\sharp_{\psi_\star}$, thus allowing the analyzer to disallow certain thread interleavings when performing state-space exploration on the generated EML program.

3.1 Implementing Random Isolation

We implemented random isolation via the source-to-source transformation outlined in §2. To keep the source-to-source transformation semantics-preserving, *and* to ensure that only one concrete object can be allocated at $\psi_\star$, an atomic "test-and-set" operation must be performed on the boolean flag G_ψ.[2] Without the use of an atomic "test-and-set", the source-to-source transformation introduces a race condition that allows multiple objects to be allocated at $\psi_\star$. This in turn would invalidate *Random-Isolation Principles 1 & 2.*

While the use of a source-to-source transformation is not strictly necessary to implement random isolation, it allows existing object-sensitive analyses to be used with minimal changes. For example, let Pts be the points-to relation computed via a flow-insensitive, object-sensitive points-to analysis in the style of [8], and CG be an object-sensitive call graph.[3] Because these two analysis artifacts are object-sensitive, their respective dataflow facts make a distinction between those for $\psi_\star$ and those for ψ. For example, if T defines a method $T.m$, then CG will contain at least two nodes for $T.m$: one for object context $\psi_\star$, and one for object context ψ. Thus, inside of the control-flow graph for $T.m$ with

[2] We use "test-and-set" to emphasize that random isolation is not particular to Java. For Java, we use the method `AtomicBoolean.compareAndSwap`.

[3] An object-sensitive call graph CG models the interprocedural control flow of a program: there is a node in CG for each method of the program for each context in which it can be invoked [8]. An object-sensitive points-to analysis associates points-to facts with the nodes of CG, thus computing different points-to facts for different object contexts of the same method.

object context $\psi_\star$, an analysis is able to take advantage of the fact that the special Java `this` variable is referring to the non-summary object $\varsigma^\sharp_{\psi_\star}$. That is, a unique context of $T.m$ has been created for $\varsigma^\sharp_{\psi_\star}$ without modifying the analyses!

In some situations, however, a CG node's context is not enough to distinguish between $\varsigma^\sharp_{\psi_\star}$ and $\varsigma^\sharp_{\psi}$. Consider the code fragment "synchronized(t) { t.m() }", where t is defined as in code fragment (2), and $\mathsf{Pts}(\mathsf{t}) = \{\varsigma^\sharp_{\psi_\star}, \varsigma^\sharp_{\psi}\}$. For performing violation detection, we require the ability to reason precisely about the state of a lock. Thus, in the program abstraction, we must be able to distinguish between the case when t references $\varsigma^\sharp_{\psi_\star}$ and when t references $\varsigma^\sharp_{\psi}$.

We solve this via a second source-to-source transformation that dispatches on the set of objects that are in Pts(t).

```
if (is_ri(t)) { synchronized(t) { t.m() } } else { synchronized(t) { t.m(); } }
```

In the source program, the method "is_ri" is defined as the identity function, and thus has no effect on the meaning of the program. However, the points-to analysis uses semantic reinterpretation of is_ri that performs a case analysis on Pts(t). Specifically, the reinterpreted is_ri performs the abstract test "$\mathsf{t} == \varsigma^\sharp_{\psi_\star}$", which allows the points-to analysis to perform *assume* statements on the branching paths (e.g., when following the true branch of the condition, the points-to analysis performs an "assume $\mathsf{Pts}(\mathsf{t}) = \{\varsigma^\sharp_{\psi_\star}\}$"). One can view this as a way to achieve object-sensitivity at the level of a program block instead of just at the method level. Although we presented this second transformation in the context of violation detection, it is a generic approach that can be applied wherever an analysis needs to distinguish between $\varsigma^\sharp_{\psi_\star}$ and $\varsigma^\sharp_{\psi}$ to perform a strong update.

4 Translation to the Empire Modeling Language (EML)

We now describe how EMPIRE defines an EML program.

4.1 Empire Modeling Language

An EML program $\mathcal{E}$ consists of (i) a finite number of shared-memory locations; (ii) a finite number of reentrant locks; and (iii) a finite number of concurrently executing processes.

An EML lock is reentrant, meaning that the lock can be reacquired by an EML process that currently owns the lock, and also that the lock must be released the same number of times to become free. EML restricts the acquisition and release of an EML lock to occur within the body of a function, i.e., an EML lock cannot be acquired in a function f and released in another function f'. In addition, the acquisition of multiple EML locks by an EML process must be properly nested: an EML process must release a set of held locks in the order opposite to their acquisition order. The two restrictions are naturally fulfilled by Java's `synchronized` blocks and methods.

Table 1. Example Java statements, their corresponding EML statements, and the condition necessary to generate the EML statement

Java	EML	Condition
`x = o.f ; o.f = x`	read m_f ; write m_f	$\varsigma^\sharp_{\psi_\star} \in \mathsf{Pts}(o)$
`sync(o){...}`	lock $\varsigma^\sharp_{\psi_\star}$;...;unlock $\varsigma^\sharp_{\psi_\star}$	$\mathsf{Pts}(o) = \{\varsigma^\sharp_{\psi_\star}\}$
`sync(o){...}`	skip;...;skip	$\mathsf{Pts}(o) \neq \{\varsigma^\sharp_{\psi_\star}\}$
`o.start()`	start P_{ψ_θ}	$P_{\psi_\theta} \in \mathsf{Pts}(o)$, `Thread.start()` invoked.
`o.u()`	unitbegin;call u; unitend	$\varsigma^\sharp_{\psi_\star} \in \mathsf{Pts}(o)$, `u()` is a unit of work
`o.m()`	call m	

An EML process is defined by a set of (possibly) recursive functions, one of which is designated as the `main` function of the process. Each function consists of a sequence of statements, each of which is either a goto, choice, skip, call f, label lab, return, read m, write m, alloc l, lock l, unlock l, unitbegin, unitend, or start P. The statement "start P" starts the EML process named P. This is used to model the fact that when a Java program begins, only one thread is executing the `main` method, and all other threads cannot begin execution until they have been started by an already executing thread. (The other kinds of statements should be self-explanatory.)

4.2 EML Generation

EMPIRE defines the EML program $\mathcal{E}$ as follows. To model the randomly-isolated abstract object $\varsigma^\sharp_{\psi_\star}$, $\mathcal{E}$ defines a shared memory location m_f for each field f of the class T, and also an EML lock $\varsigma^\sharp_{\psi_\star}$ to model the lock associated with $\varsigma^\sharp_{\psi_\star}$. The status of the global flag G_ψ is modeled by the EML lock $\varsigma^\sharp_{\psi_\star}$ being allocated or not. Let Threads be the set of all subclasses of `java.lang.Thread`. For each $\theta \in \mathsf{Threads}$, and for each allocation site ψ_θ that allocates an instance of θ, $\mathcal{E}$ defines an EML process P_{ψ_θ} that models the behavior of one instance of θ that is allocated at ψ_θ. Finally, $\mathcal{E}$ defines an EML process P_{main} that models the Java thread that begins execution of the `main` method. Each EML process P defines a function for each method that is reachable from P's entry point in CG. The translation from Java statements to EML statements is straightforward, with example translations given in Tab. 1.

5 Translation to Communicating Pushdown Systems

In this section, we describe the translation of EML programs into CPDSs.

5.1 Communicating Pushdown Systems

Definition 1. *A pushdown system (PDS) is a four-tuple* $\mathcal{P} = (Q, Act, \Gamma, \Delta)$, *where* Q *is a finite set of states,* Act *is a finite set of* actions, Γ *is a finite*

Table 2. The encoding of a call graph's and CFG's edges as PDS rules. The action a denotes the abstract behavior of executing that edge.

Rule	Control flow modeled
$\langle q, n_1\rangle \overset{a}{\hookrightarrow} \langle q, n_2\rangle$	Intraprocedural edge $n_1 \to n_2$
$\langle q, c\rangle \overset{a}{\hookrightarrow} \langle q, e_f\ r\rangle$	Call to f from c that returns to r
$\langle q, x_f\rangle \overset{a}{\hookrightarrow} \langle q, \varepsilon\rangle$	Return from f at exit node x_f

stack alphabet, and Δ is a finite set of rules of the form $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', u'\rangle$, where $q, q' \in Q$, $a \in Act$, $\gamma \in \Gamma$, and $u' \in \Gamma^*$. A configuration of $\mathcal{P}$ is a pair $c = \langle q, u\rangle$, where $q \in Q$ and $u \in \Gamma^*$ is the stack contents. A set of configurations C is regular if for each $q \in Q$ the language $\{u \in \Gamma^* \mid \langle q, u\rangle \in C\}$ is regular.

We assume that associated with each PDS $\mathcal{P}$ is an initial configuration c_{init}. For all $u \in \Gamma^*$, a configuration $c = \langle q, \gamma u\rangle$ can make a transition to a configuration $c' = \langle q', u'\, u\rangle$ if there exists a rule $r \in \Delta$ of the form $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', u'\rangle$. We denote this transition by $\overset{a}{\longrightarrow}$ and extend it to $\overset{a_1 \cdots a_n}{\longrightarrow}$ in the obvious manner. For a set of configurations C, we define the target language of $\mathcal{P}$ with respect to C as $\mathsf{Lang}(\mathcal{P}, C) = \{w \mid \exists c \in C, w \in Act^*, c_{\mathsf{init}} \overset{w}{\longrightarrow} c\}$.

Because PDSs maintain a stack, they naturally model the interprocedural control flow of a thread of execution. The translation from a call graph and set of control-flow graphs (CFGs) into a PDS is shown in Tab. 2.

Definition 2. *A communicating pushdown system (CPDS) is a tuple $\mathsf{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ of PDSs. The action set Act of CP is equal to the union of the action sets of the $\mathcal{P}_i$, along with the special action τ: τ has the property that for all $a \in \bigcup_1^n Act_i$, $\tau a = a\tau = a$. The rules Δ_i for PDS $\mathcal{P}_i$ are augmented to include $\{\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q, \gamma\rangle \mid q \in Q_i, \gamma \in \Gamma_i, a \in (Act \setminus Act_i)\}$.*

Given n sets of configurations $S = (C_1, \ldots, C_n)$, we define the target language of a CPDS CP with respect to S as $\mathsf{Lang}(\mathsf{CP}, S) = \bigcap_{1 \le i \le n} \mathsf{Lang}(\mathcal{P}_i, C_i)$, where intersection enforces that all the $\mathcal{P}_i$ synchronize on the global actions. The goal of the CPDS model checker [3] is to determine if $\mathsf{Lang}(\mathsf{CP}, S)$ is empty. Because each language $\mathsf{Lang}(\mathcal{P}_i, C_i)$ can be, in general, a context-free language, and the problem of checking their intersection for emptiness is known to be undecidable, the CPDS model checker algorithm is only a *semi-decision* procedure. The semi-decision procedure may not terminate, but is guaranteed to terminate if there exists a finite-length sequence of actions, $w = a_1 \ldots a_n$, such that $w \in \mathsf{Lang}(\mathsf{CP}, S)$. Additionally, in some cases, the semi-decision procedure can determine that $\mathsf{Lang}(\mathsf{CP}, S) = \emptyset$. We refer the reader to [3] for more details.

5.2 CPDS Generation

An EML program has a set of shared-memory locations, S_{Mem}, a set of EML locks, S_{Locks}, and a set of EML processes, S_{Procs}. Empire generates a number of

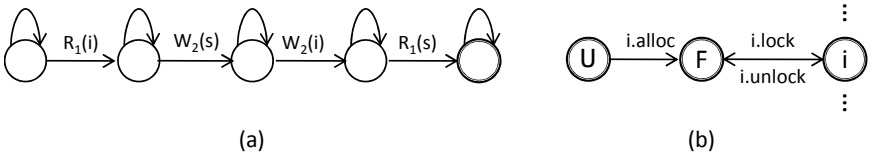


Fig. 2. (a) Race automaton for interleaving scenario 12 for example program in Fig. 1. (b) Lock automaton template. There is a state "i" for each EML process in S_{Procs}.

CPDSs for a given EML program: a CPDS is generated for each pair $(m_f, m_g) \in S_{\mathsf{Mem}} \times S_{\mathsf{Mem}}$ for the fourteen interleaving scenarios. Pairs are used because the interleaving scenarios are defined in terms of at most two locations from an atomic set [4]. In total, EMPIRE generates $O(14 * (|S_{\mathsf{Mem}}|^2))$ CPDSs for an EML program.

For a generated CPDS CP, there is a PDS for each global component of the EML program: CP contains a PDS that monitors for a violation, a PDS for each lock, and a PDS for each EML process. We now describe the generation technique for each component in turn. When the target language of a PDS is regular, we define it in terms of a finite-state machine (FSM). (An FSM is a single-state PDS with no push or pop rules; the initial configuration describes the initial state; and the final set of configurations describes the accepting state(s) of the FSM.)

The *violation monitor* detects when one of the interleaving scenarios occurs during a unit of work. The violation monitor is defined by a *race automaton* [4], which is a finite automaton that contains one state for each access defined by the scenario; transitions between states that reflect that an access has occurred; and self-transitions on states for accesses that do not make the scenario progress. Fig. 2(a) shows the race automaton that accepts the violation of scenario 12 for the example program.

Because an EML lock is reentrant, the language of the PDS that describes such behavior is context-free. However, previous work by the authors [9] developed a technique that safely removes reentrant acquisitions from an EML process, enabling the EML lock to be modeled as an FSM. Fig. 2(b) depicts a template FSM for one EML lock. The FSM begins in the Unallocated state, transitions to the Free state upon being allocated, and alternates between an "acquired-by-process-i" state and the Free state. Transitioning from Unallocated to Free denotes setting the global flag G_ψ associated with $\varsigma^\sharp_{\psi_\star}$.

Generating a PDS $\mathcal{P}$ for an EML process P is performed in two stages. First, a single-state PDS $\mathcal{P}_1 = (Q_1, Act_1, \Gamma_1, \Delta_1)$ is generated using the rule templates depicted in Tab. 2, with Act_1 being the set of all distinct EML statements used by P. $\mathcal{P}_1$ captures the interprocedural control flow of P.

Second, PDS $\mathcal{P}_2 = (Q_2, Act_2, \Gamma_2, \Delta_2)$ is defined as follows: $Q_2 = 2^{S_{\mathsf{Locks}}} \times Q_1$, $Act_2 = \{P.a \mid a \in (Act_1 \setminus \mathsf{start})\} \cup \{P'.\mathsf{alloc}\ \varsigma^\sharp_\star \mid P' \in (S_{\mathsf{Procs}} \setminus \{P\})\}$, $\Gamma_2 = \Gamma_1 \cup \{\mathsf{guess}\}$, and Δ_2 is defined from Δ_1 as shown in Tab. 3. Attaching the EML process's name P to the actions in Act_2 enables the violation monitor and locks to know which EML process performs an action. In Tab. 3, row 2 ensures that no lock is allocated more than once; row 3 ensures that a lock is not used before

Table 3. Each row defines a set of PDS rules that are necessary for modeling the allocation of locks (see §5.2)

Action a	Rule $\langle q,\gamma\rangle \xhookrightarrow{a} \langle q',w\rangle$	
τ, start $\mathcal{P}'$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}}\ \}$
alloc $\varsigma^\sharp_\star$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s',p'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_\star \notin s \wedge s' = s \cup \{\varsigma^\sharp_\star\}\ \}$
lock/unlock $\varsigma^\sharp_\star$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_\star \in s\ \}$
read/write m_f	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_{\psi_\star} \in s\ \}$
ubegin/uend	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_{\psi_\star} \in s\ \}$
$\star$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{\tau} \langle (s,q),\mathsf{guess}\ \gamma\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}}\ \}$
$\star$	$\{\ \langle (s,q),\ \mathsf{guess}\rangle \xhookrightarrow{P'.\mathsf{alloc}\ \varsigma^\sharp_\star} \langle (s',p),\epsilon\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_\star \notin s \wedge s' = s \cup \{\varsigma^\sharp_\star\} \wedge P' \in (S_{\mathsf{Procs}} \setminus \{P\})\ \}$

being allocated; and rows 4 and 5 ensure that the shared-memory locations are not accessed before $\varsigma^\sharp_{\psi_\star}$ has been allocated. Row 6 defines rules that invoke the "guessing" procedure for each configuration of $\mathcal{P}_2$. Guessing is necessary because an EML process cannot know when another EML process allocates a lock. Row 7 defines rules that implement the guessing procedure: from state (s,q), $s \subseteq S_{\mathsf{Locks}}$, guess that EML process $P' \in (S_{\mathsf{Procs}} \setminus \{P\})$ allocates a lock $\varsigma^\sharp_\star \in (S_{\mathsf{Locks}} \setminus s)$, and return back to the caller in the new state $(s \cup \{\varsigma^\sharp_\star\}, q)$. The guessing rule is then labeled with action $P'.\mathsf{alloc}\ \varsigma^\sharp_\star$.

Once CP has been generated, a language-emptiness query is passed to the CPDS model checker. This requires defining the target set of configurations for each PDS $\mathcal{P}_i$. For a PDS whose target language is regular, the target set of configurations is defined by the FSM. For a PDS that describes an EML process, the target set of configurations is any configuration (i.e., $\{\langle q,u\rangle \mid q \in Q, u \in \Gamma^*\}$). Let S be the configuration sets for the PDSs. The language-emptiness query as defined is such that $\mathsf{Lang}(\mathsf{CP},S) = \emptyset$ is true *if-and-only-if* the EML program cannot generate a trace accepted by the violation monitor.

6 Experiments

Empire is implemented using the WALA [10] program-analysis framework. Random isolation uses WALA's support for rewriting the abstract-syntax tree of a Java program. The default object-senstive call graph construction and points-to analyses are modified to implement the semantic reinterpretation of "is_ri", as described in §3.1.

We evaluated Empire on eight programs from the ConTest suite [6], which is a set of small benchmarks with known non-trivial concurrency bugs. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 16 GB of memory. The analyzed programs are modified versions of those in the ConTest suite. To reduce the size of the generated models, we removed all use of

Table 4. Marked entries denote violations reported by EMPIRE, with ✓ being a verified violation and ✓̸ a false positive. Scenarios 6-11 involve two memory locations.

Nr	Program	CPDSs	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	account	352	✓	✓		✓	✓									
2	airlinesTckts	630	✓	✓		✓		✓	✓	✓						
3	AllocationV	15	✓	✓												
4	BuggyProgram	68				✓										
5	BugTester	435														
6	PingPong	460	✓	✓	✓		✓	✓̸	✓̸	✓̸	✓̸	✓̸	✓̸		✓̸	✓̸
7	ProdConsumer	291	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓
8	Shop	542	✓	✓		✓							✓	✓	✓	✓
	Totals	2793														

file I/O from the programs. When a benchmark used a shared object of type `java.lang.Object` as a lock, the type was changed to `java.lang.Integer` because our implementation uses *selective* object-sensitivity, for which the use of `java.lang.Object` as a shared lock removes all selectivity and severely degrades performance. The programs `AllocationV` and `Shop` define a thread's `run()` method that consists of a loop that repeatedly executes one unit of work. For these programs, the code body of the loop was extracted out into its own method so that the default unit-of-work assumptions would be correct. Finally, many benchmarks allocate threads in a loop. We manually unrolled these loops to make the programs use a finite number of threads.

For 6 of the 8 benchmarks listed in Tab. 4, EMPIRE found multiple violations. The false positives reported for `PingPong` are due to an overapproximation of a thread's control flow—exceptional control paths are allowed in the model that cannot occur during a real execution of the program. The program `ProducerConsumer` has an atomic set with mutiple fields and uses no synchronization. While not interesting for violation detection, it validates that our approach is able to detect each of the problematic interleaving scenarios. Overall, the initial results are encouraging for applying EMPIRE to larger programs. Future work on EMPIRE includes a thread-escape analysis—determining the allocation sites that allocate shared objects—which would allow EMPIRE to analyze the escaping allocation sites using the default assumptions.

7 Related Work

Strong updates on an isolated non-summary object. The idea of isolating a distinguished non-summary node that represents the memory location that will be updated during a transition, so that a strong update can be performed on it, has a long history in shape-analysis algorithms [11,12,13]. When these methods also employ the allocation-site abstraction, each abstract memory configuration will have some bounded number of abstract nodes per allocation site.

Like random-isolation abstraction, *recency abstraction* [14] uses no more than *two* abstract blocks per allocation site ψ: a non-summary block MRAB[ψ], which

represents the **m**ost-**r**ecently-**a**llocated **b**lock allocated at ψ, and a summary block NMRAB[ψ], which represents the **n**on-**m**ost-**r**ecently-**a**llocated **b**locks allocated at ψ. As the names indicate, recency abstraction is based on tracking a *temporal* property of a block b: the is-the-most-recent-block-from-$\psi(b)$ property.

With counter abstraction [15,16,17], numeric information is attached to summary objects to characterize the number of concrete objects represented. The information on summary object u of abstract configuration S describes the number of concrete objects that are mapped to u in any concrete configuration that S represents. Counter abstraction has been used to analyze infinite-state systems [15,16], as well as in shape analysis [17].

In contrast to all of the aforementioned work, random-isolation abstraction is based on tracking the properties of a *random* individual, and generalizing from the properties of the randomly chosen individual according to Random-Isolation Principle 2.

Detection of concurrency-related bugs. Traditional work on error detection for concurrent programs has focused on classical data races. Static approaches for detecting data races include type systems, where the programmer indicates proper synchronization via type annotations (see e.g., [18]), model checking (see e.g., [19]), and static analysis (see e.g., [20]). Dynamic analyses for detecting data races include those based on the lockset algorithm [21], on the happens-before relation [22], or on a combination of the two [23]. A data race is a heuristic indication that a concurrency bug may exist, and does not directly correspond to a notion of program correctness. In our approach, we consider atomic-set serializability as a correctness criterion, which captures the programmer's intentions for correct behavior directly.

High-level data races may take the form of *view inconsistency* [24], where memory is read inconsistently, as well as *stale-value errors* [25], where a value read from a shared variable is used beyond the synchronization scope in which it was acquired. Our problematic interleaving scenarios capture these forms of high-level data races, as well as several others, in one framework.

Several notions of serializability (or atomicity) and associated detection tools have been presented, including [26,27,28,29]. These correctness criteria ignore relationships that may exist between shared memory locations, and treat all locations as forming one atomic set. Therefore, they may not accurately reflect the intentions of the programmer for correct behavior. Atomic-set-serializability takes such relationships into account and provides a finer-grained correctness criterion for concurrent systems. For a detailed discussion and comparison of different notions of serializability see [4].

Atomic-set serializality was proposed by Vaziri et al. [1]. That work focused on inference of locks. A dynamic violation-detection tool was proposed in [4] to find errors in legacy code. Our tool is a static counterpart with the benefit that it (symbolically) considers multiple executions of a program, instead of just one execution like the dynamic tool.

References

1. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL (2006)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL (2003)
3. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
4. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE (2008)
5. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. In: TSE (1986)
6. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Conc. and Comp.: Prac. and Exp. 19(3), 267–279 (2007)
7. Jones, N., Muchnick, S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: POPL (1982)
8. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. TOSEM 14(1) (2005)
9. Kidd, N., Lal, A., Reps, T.: Language strength reduction. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 283–298. Springer, Heidelberg (2008)
10. Watson Libraries for Analysis (WALA), T.J.: `http://wala.sourceforge.net/wiki/index.php`
11. Horwitz, S., Pfeiffer, P., Reps, T.: Dependence analysis for pointer variables. In: PLDI (1989)
12. Jones, N., Muchnick, S.: Flow analysis and optimization of Lisp-like structures. In: Program Flow Analysis: Theory and Applications. Prentice-Hall, Englewood Cliffs (1981)
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3) (2002)
14. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
15. McMillan, K.: Verification of infinite state systems by compositional model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 219–237. Springer, Heidelberg (1999)
16. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0, 1, \infty)$-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 107. Springer, Heidelberg (2002)
17. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 69–84. Springer, Heidelberg (2002)
18. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA (2002)
19. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI (2004)
20. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL (2007)
21. von Praun, C., Gross, T.R.: Object race detection. In: OOPSLA (2001)
22. Min, S.L., Choi, J.D.: An efficient cache-based access anomaly detection scheme. In: ASPLOS (1991)

23. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: PPoPP (2003)
24. Artho, C., Havelund, K., Biere, A.: High-level data races. In: Proc. NDDL/VVEIS 2003 (2003)
25. Burrows, M., Leino, K.R.M.: Finding stale-value errors in concurrent programs. Conc. and Comp.: Prac. and Exp. 16(12) (2004)
26. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multi-threaded programs. In: POPL, pp. 256–267 (2004)
27. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analysis of data races and atomicity. In: PPoPP (2005)
28. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting atomicity violations via access interleaving invariants. In: ASPLOS (2006)
29. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP (2006)

An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries

Johannes Kinder[1], Florian Zuleger[1,2,⋆], and Helmut Veith[1]

[1] Technische Universität Darmstadt, 64289 Darmstadt, Germany
[2] Technische Universität München, 85748 Garching, Germany

Abstract. Due to indirect branch instructions, analyses on executables commonly suffer from the problem that a complete control flow graph of the program is not available. Data flow analysis has been proposed before to statically determine branch targets in many cases, yet a generic strategy without assumptions on compiler idioms or debug information is lacking.

We have devised an abstract interpretation-based framework for generic low level programs with indirect jumps which safely combines a pluggable abstract domain with the notion of partial control flow graphs. Using our framework, we are able to show that the control flow reconstruction algorithm of our disassembly tool Jakstab produces the most precise overapproximation of the control flow graph with respect to the used abstract domain.

1 Introduction

One of the main problems when analyzing low level code, such as x86 assembly language, are indirect branch instructions. These correspond to goto statements where the target is calculated at runtime, or the use of function pointers combined with pointer arithmetic in high level languages. In executables, any address in the code is a potential target of an indirect branch, since in general there are no explicit labels. Failure to statically resolve the target of an indirect branch instruction thus leads to an either incomplete or grossly overapproximated control flow graph. Often, data flow analysis can aid in resolving such indirect branches; however, data flow analysis already requires a precise control flow graph to work on. This seemingly paradox situation has been referred to as an inherent "chicken and egg" problem in the literature [1,2].

In this paper, we show that this notion is overly pessimistic. We present a framework to construct a safe overapproximation of the control flow graph of low level programs by effectively combining control and data flow analysis by means of abstract interpretation. Existing approaches to control flow extraction from binaries usually either make a best effort attempt and accept possible unsoundness [3,4], or they make optimistic assumptions on clean code layout [5] or on the presence of additional information such as symbol tables or relocation information [2]. Our approach is designed to be generic in the sense that it does not require any additional information besides the actual instructions and is still able to compute a sound and precise overapproximation of the control flow graph. In particular, our paper makes the following contributions:

⋆ The research of the second author was supported in part by Microsoft Research through its PhD Scholarship Programme.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 214–228, 2009.

- We define an abstract interpretation that reconstructs the control flow graph and is parameterized by a given abstract domain. We achieve this by extending the given abstract domain with a representation of the partial control flow graph. To this end, we define the notion of a control flow graph for a low level assembly-like language based on a concrete semantics in Section 3.2. We construct a *resolve* operator, based on conditions imposed on the provided abstract domain, for calculating branch targets. Using this operator, our analysis is able to safely overapproximate the control flow graph (Section 3.3).
- We present a general extension of the classical worklist algorithm met in program analysis which empowers control flow reconstruction by data flow analyses under very general assumptions. The algorithm overcomes the "chicken and egg" problem by computing the a priori unknown edges on the fly by using the *resolve* operator. We prove that the algorithm always returns the most precise overapproximation of the program's actual control flow graph with respect to the precision of the provided abstract domain used by the data flow analysis (Section 3.4).
- In earlier work, we presented our disassembly tool JAKSTAB [6], which employs constant propagation for an iterative disassembly strategy. JAKSTAB uses an abstract domain which supports symbolic memory addresses to achieve constant propagation through local variables and yielded better results than the most widely used commercial disassembler IDA Pro. We describe in Section 4 how the control flow reconstruction algorithm implemented in JAKSTAB instantiates our newly defined abstract interpretation. Thus, without the need to restart constant propagation, it always computes a safe overapproximation of the control flow graph.

2 Overview

In this section we describe current disassembly techniques and their shortcomings. We explain why proposed augmentations of disassembly with data flow analysis suffer from imprecision and we motivate how to overcome these difficulties by an intertwined control and data flow analysis.

2.1 Disassembly

Disassembly is the process of translating a sequence of bytes into an assembly language program. Simple *linear sweep* disassemblers, such as GNU objdump, sequentially map all bytes to instructions. Especially on architectures with varying instruction length (e.g. Intel x86) this leads to erroneous assembly programs, as these disassemblers easily lose the correct alignment of instructions because of data or padding bytes between code blocks. *Recursive traversal* disassemblers interpret branch instructions in the program to translate only those bytes which can actually be reached by control flow. The disassembler, however, cannot always determine the target of a branch instruction and can thus miss parts of the program.

To avoid this problem, disassemblers usually augment recursive traversal by heuristics to detect potential pieces of code in the executable. These heuristics exploit the presence of known compiler idioms, such as recurring procedure prologues or common

patterns in the calculation of switch-jumps from jump tables [4]. While this works well for the majority of compiler generated code, the presence of hand-crafted assembly code and the effects of aggressive compiler optimizations can thwart heuristic methods. Moreover, heuristics in disassembly are prone to creating false positives, i.e., to misinterpret data as code. Because of these well known problems, improved methods of disassembly that incorporate data flow analysis have been subject to research.

2.2 Augmenting Disassembly with Data Flow Analysis

Data flow analysis statically calculates information about the program variables from a given program representation. Earlier work [1,5,3,7,6] has shown that data flow analysis can be used to augment the results of disassembly, but no conclusive answer was given on the best way to handle states with unresolved control flow successors during data flow analysis. Further, updating the control flow graph could render previous data flow information invalid, which would require backtracking and could cause the analysis to diverge.

De Sutter et al. [7] suggested to initially connect all indirect jumps to a virtual *unknown* node for indirect jumps, which effectively overapproximates the control flow graph. In an iterative process, they use constant propagation on the overapproximated graph to show infeasibility of most overapproximated edges, which can then be removed. This approach is inspired by the solution of Chang et al. [8] to the similar problem of treating unknown external library functions in the analysis of C programs. We exemplify De Sutter et al.'s method by applying it to the snippet of pseudo-assembly code shown in the left of Figure 1. The center of the figure depicts the corresponding initial control flow graph, where the indirect jump at line 2 is connected to the unknown node ($\top$). There are outgoing edges from the unknown node to all statements, since every address is a potential jump target in the general case of stripped code without relocation information. Calculating the possible values of x, we see that x can in fact take the concrete values $5, 3, 1, -1, \ldots$ at the entry of line 2 in the overapproximated program. Thus a program analysis operating on this initial overapproximation can only conclude that addresses 2 and 4 are no targets of the jump, but cannot remove the overapproximated edges to addresses 1 and 3. The final CFG reconstructed by this method,

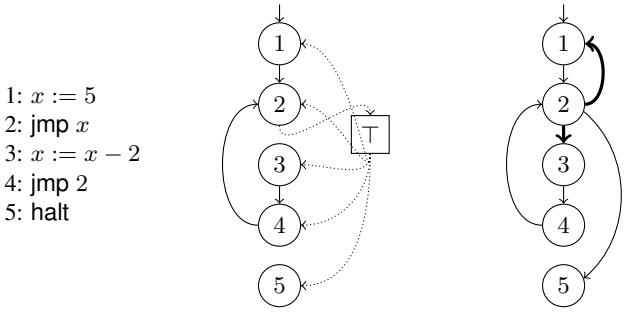


Fig. 1. Overapproximation of the CFG by adding an unknown node ($\top$) leads to additional possible values for x at the indirect jump

shown on the right of Figure 1, consequently contains the infeasible edges (2,1) and (2,3) (drawn in bold).

This example shows that integration of data flow algorithms with control flow reconstruction is non-trivial and can lead to suboptimal results. In the rest of this paper we will demonstrate how to successfully design program analyses which reconstruct the control flow of a disassembled low level program.

2.3 Integrating Fixed Point Equations for Control and Data Flow Analysis

Our approach to control flow reconstruction is based on the idea of executing data flow analysis and branch resolution simultaneously. A data flow problem is characterized by a constraint system derived from an overapproximation of the program semantics. The solution to a data flow problem is calculated by iteratively applying the constraints until a fixed point is reached. These constraints encode the complete control flow graph (by an edge relation), which, however, is unavailable as our task exactly is the control flow reconstruction.

The intuition of our approach is that we can grow the edge relation during the fixed point iteration, until a simultaneous least fixed point of both data and control flow is reached. For growing the edge relation, we introduce a $\mathbf{resolve}$ operator that uses data flow information to calculate branch targets of instructions. In our combined analysis, we will ensure that

- the quality of the fixed point, and thus of the reconstructed control flow graph, does not depend on the order in which the constraints are applied.
- the fixed point of control and data flow is a valid abstraction of the concrete program behavior.

3 Abstract Interpretation of Low Level Languages

In this section we formally define our combined analysis and prove the above properties. First, we introduce our low level language, then its concrete semantics, and finally we state our abstract interpretation framework. Our notation follows standard program analysis literature [9].

3.1 A Simple Low Level Language

We restrict ourselves to a simple low level language, JUMP, which captures the specifics of assembly language. JUMP uses the *program counter* pc, a finite set of *integer variables* $V = \{v_1, \ldots, v_n\}$, and a *store* $m[\cdot]$. For generality we do not further specify the *expressions* in JUMP, even though we explicitly note that expressions may contain the program counter, the variables and the store. We denote the set of expressions by $\mathbf{Exp}$. A *statement* in JUMP can be either

- a *variable assignment* $v := e$, where $v \in V$ and $e \in \mathbf{Exp}$, which assigns the value of an expression e to the variable v,
- a *store assignment* $m[e_1] := e_2$, where $e_1, e_2 \in \mathbf{Exp}$, which assigns the value of e_2 to the store location specified by e_1,

- a *guarded jump* statement $\mathsf{jmp}\ e_1, e_2$, with $e_1, e_2 \in \mathbf{Exp}$, which transfers control to the statement at the address calculated from expression e_2 if the expression e_1 evaluates to 0,
- or the *program exit* statement halt, which terminates execution of the program.

We denote the set of statements by $\mathbf{Stmt}$. The set of program *addresses* $\mathbf{A} \subseteq \mathbb{Z}$ is a *finite* subset of the integers. A program in JUMP is a finite partial mapping of addresses to statements. The idea is that every program has a fixed finite representation. At first we will assume that all addresses in $\mathbf{A}$ correspond to statements of the program. After we finish control flow reconstruction, we establish that some statements are not reachable and we can conclude that they are not part of the program (e.g., pieces of data intermixed with code blocks or misaligned statements on architectures with variable length instructions). Every program in our language JUMP has a unique starting address $\mathbf{start}$. The mapping between addresses and statements is expressed by $[stmt]^a$, where $stmt \in \mathbf{Stmt}$ and $a \in \mathbf{A}$. We present the formal semantics of JUMP in the next section.

JUMP is representative for assembly languages, since the most problematic features of machine code, indirect jumps and indirect addressing, are fully supported. Intuitively, it forms a minimalist intermediate representation for machine code. For simplicity JUMP does not implement explicit call and return instructions as these can be implemented by storing the program counter and jumping to the procedure, and jumping back to the stored value, respectively.

3.2 Semantics of JUMP

The semantics of JUMP is defined in terms of *states*. The set of states $\mathbf{State} := \mathbf{Loc} \times \mathbf{Val} \times \mathbf{Store}$ is the product of the *location valuations* $\mathbf{Loc} := \{pc\} \to \mathbf{A}$, the *variable valuations* $\mathbf{Val} := V \to \mathbb{Z}$ and the *store valuations* $\mathbf{Store} := \mathbb{Z} \to \mathbb{Z}$. We refer to the part of a state that represents an element of $\mathbf{Store}$ by a function $m[\cdot]$. As a state s is a function, we denote by $s(pc)$ the value of the program counter, by $s(v_i)$ the value of a variable v_i, and by $s(m[c])$ the value of the store mapping for an integer $c \in \mathbb{Z}$. We denote by $s[\cdot \mapsto \cdot]$ the state we obtain after substituting a new value for either the program counter, a variable, or a store mapping in s. We assume the existence of a deterministic *evaluation* function $\mathbf{eval} : \mathbf{Exp} \to \mathbf{State} \to \mathbb{Z}$ ($\to$ is right-associative, i.e., $\mathbf{Exp} \to \mathbf{State} \to \mathbb{Z}$ stands for $\mathbf{Exp} \to (\mathbf{State} \to \mathbb{Z})$). We now define the operator $\mathbf{post} : \mathbf{Stmt} \to \mathbf{State} \to \mathbf{State}$:

$$
\begin{aligned}
\mathbf{post}[\![v := e]\!](s) &:= s[v \mapsto \mathbf{eval}[\![e]\!](s)][pc \mapsto s(pc) + 1] \\
\mathbf{post}[\![m[e_1] := e_2]\!](s) &:= s[m[\mathbf{eval}[\![e_1]\!](s)] \mapsto \mathbf{eval}[\![e_2]\!](s)][pc \mapsto s(pc) + 1] \\
\mathbf{post}[\![\mathsf{jmp}\ e_1, e_2]\!](s) &:= \begin{cases} s[pc \mapsto \mathbf{eval}[\![e_2]\!](s)] & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto s(pc) + 1] & \text{otherwise} \end{cases}
\end{aligned}
$$

Remark 1. For the ease of explanation we have chosen to assume that all statements are of length 1, and therefore the program counter is increased by 1 for fall-through edges. Note that it would make no conceptual difference to introduce a length function that calculates the appropriate next location for every statement.

For later use in the definition of control flow graphs and in control flow reconstruction we define a language $\mathbf{Stmt}^{\#}$ derived from the language $\mathbf{Stmt}$, which consists of *assignments* $v := e$, $m[e_1] := e_2$ and labeled *assume* statements $\mathsf{assume}_a(e = 0)$, $\mathsf{assume}_a(e \neq 0)$, where $e, e_1, e_2 \in \mathbf{Exp}, a \in \mathbf{A}$, but which does not contain guarded jump statements. The intuition is that the assume statements correspond to resolved jump statements of the language $\mathbf{Stmt}$, where the labels specify resolved target addresses of the jump statements. We overload the operator $\mathbf{post} : \mathbf{Stmt}^{\#} \to \mathbf{2^{State}} \to \mathbf{2^{State}}$ to work on statements of the derived language and sets of states $S \subseteq \mathbf{State}$:

$$\begin{aligned}
\mathbf{post}[\![v := e]\!](S) &:= \{\mathbf{post}[\![v := e]\!](s) \mid s \in S\},\\
\mathbf{post}[\![m[e_1] := e_2]\!](S) &:= \{\mathbf{post}[\![m[e_1] := e_2]\!](s) \mid s \in S\},\\
\mathbf{post}[\![\mathsf{assume}_a(e = 0)]\!](S) &:= \{s[pc \mapsto a] \mid \mathbf{eval}[\![e]\!](s) = 0, s \in S\},\\
\mathbf{post}[\![\mathsf{assume}_a(e \neq 0)]\!](S) &:= \{s[pc \mapsto a] \mid \mathbf{eval}[\![e]\!](s) \neq 0, s \in S\}.
\end{aligned}$$

Note that the definition of the $\mathbf{post}$ operator over sets makes use of the $\mathbf{post}$ operator for single elements in the case of assignments. We will need $\mathbf{Stmt}^{\#}$ and the transfer function $\mathbf{post}$ when stating the conditions we require from the abstract domain for our control flow reconstruction in Section 3.3.

A *trace* σ of a program is a finite sequence of states $(s_i)_{0 \leq i \leq n}$, such that $s_0(pc) = \mathbf{start}$, $stmt$ is not halt for all $[stmt]^{s_i(pc)}$ with $0 \leq i < n$, and $s_{i+1} = \mathbf{post}[\![stmt]\!](s_i)$ for all $[stmt]^{s_i(pc)}$ with $0 \leq i < n$. Note that we do not impose conditions on variable or store valuations for state s_0. We denote the set of all traces of a program by $\mathbf{Traces}$. Further, we assume the program counter of all states in all traces to only map into the finite set of addresses $\mathbf{A}$, as every program has a fixed finite representation.

The definition of control flow graphs of programs in JUMP is based on our definition of traces and uses labeled edges. We define the set of labeled edges $\mathbf{Edge}$ to be $\mathbf{A} \times \mathbf{Stmt}^{\#} \times \mathbf{A}$.

Definition 1 ((Trace) Control Flow Graph). *Given a trace $\sigma = (s_i)_{0 \leq i \leq n}$, the trace control flow graph (TCFG) of σ is*

$$\begin{aligned}
TCFG(\sigma) = \{&(s_i(pc), stmt, s_{i+1}(pc)) \mid\\
&0 \leq i < n \text{ with } [stmt]^{s_i(pc)}, \text{ where } stmt \text{ is } v := e \text{ or } m[e_1] := e_2\}\\
\cup \{&(s_i(pc), \mathsf{assume}_{s_{i+1}(pc)}(e_1 = 0), s_{i+1}(pc)) \mid\\
&0 \leq i < n \text{ with } [\mathsf{jmp}\ e_1, e_2]^{s_i(pc)} \text{ and } \mathbf{eval}[\![e_1]\!](s_i) = 0\}\\
\cup \{&(s_i(pc), \mathsf{assume}_{s_{i+1}(pc)}(e_1 \neq 0), s_{i+1}(pc)) \mid\\
&0 \leq i < n \text{ with } [\mathsf{jmp}\ e_1, e_2]^{s_i(pc)} \text{ and } \mathbf{eval}[\![e_1]\!](s_i) \neq 0\}.
\end{aligned}$$

The control flow graph (CFG) is the union of the TCFGs of all traces

$$CFG = \bigcup_{\sigma \in \mathbf{Traces}} TCFG(\sigma).$$

As stated in the above definition, the CFG of a program is a semantic property, not a syntactic one, because it depends on the possible executions.

3.3 Control Flow Reconstruction by Abstract Interpretation

For the purpose of CFG reconstruction we are interested in abstract domains $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$, where

- $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq)$ is a complete lattice,
- the *concretization* function $\gamma : L \to \mathbf{2}^{\mathbf{State}}$ is monotone, i.e.,

$$\ell_1 \sqsubseteq \ell_2 \Rightarrow \gamma(\ell_1) \subseteq \gamma(\ell_2) \quad \text{for all } \ell_1, \ell_2 \in L,$$

  and maps the least element to the empty set, i.e., $\gamma(\bot) = \emptyset$,
- the *abstract operator* $\widehat{\mathbf{post}} : \mathbf{Stmt}^{\#} \to L \to L$ overapproximates the concrete transfer function $\mathbf{post}$, i.e.,

$$\mathbf{post}[\![stmt]\!](\gamma(\ell)) \subseteq \gamma(\widehat{\mathbf{post}}[\![stmt]\!](\ell)) \quad \text{for all } stmt \in \mathbf{Stmt}^{\#}, \ell \in L, \text{ and}$$

- the *abstract evaluation* function $\widehat{\mathbf{eval}} : \mathbf{Exp} \to L \to L$ overapproximates the concrete evaluation function, i.e.,

$$\mathbf{eval}[\![e]\!](\gamma(\ell)) \subseteq \gamma(\widehat{\mathbf{eval}}[\![e]\!](\ell)) \quad \text{for all } e \in \mathbf{Exp}, \ell \in L.$$

In the following we define a control flow analysis based on an abstract domain $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$. Our control flow analysis works on a *Cartesian abstract domain* $D : \mathbf{A} \to L$ and a *partial control flow graph* $F \subseteq \mathbf{Edge}$. The fact that edges are labeled with statements from $\mathbf{Stmt}^{\#}$ enables us to combine the abstract domain with the control flow reconstruction nicely.

A control flow analysis must have the ability to detect the (possibly overapproximated) set of targets of guarded jumps based on the knowledge it acquires. To this end, we define the operator $\mathbf{resolve} : \mathbf{A} \to L \to \mathbf{2}^{\mathbf{Edge}}$, using the functions available in the abstract domain. For a given address a and a lattice element ℓ, $\mathbf{resolve}$ returns a set of labeled control flow graph edges. If ℓ is the least element $\bot$, the location a has not been reached by the abstract interpretation yet, therefore no edge needs to be created and the empty set is returned. Otherwise, $\mathbf{resolve}$ labels fall-through edges with their respective source statements, or it calculates the targets of guarded jumps based on the information gained from the lattice element ℓ and labels the determined edges with their respective conditions:

$$\mathbf{resolve}_a(\ell) :=$$
$$:= \begin{cases} \emptyset & \text{if } \ell = \bot \\ \{(a, stmt, a+1)\} & \text{if } \ell \neq \bot \text{ and } ([stmt]^a \text{ is } [v := e]^a \\ & \qquad \text{or } [m[e_1] := e_2]^a) \\ \{(a, \mathsf{assume}_{a'}(e_1 = 0), a') \mid & \\ \quad a' \in \gamma\Big(\widehat{\mathbf{eval}}[\![e_2]\!]\big(\widehat{\mathbf{post}}[\![\mathsf{assume}_a(e_1 = 0)]\!](\ell)\big)\Big) \cap \mathbf{A}\} & \\ \quad \cup \{(a, \mathsf{assume}_{a+1}(e_1 \neq 0), a+1)\} & \text{if } \ell \neq \bot \text{ and } [\mathsf{jmp}\ e_1, e_2]^a \end{cases}$$

The crucial part in the definition of the $\mathbf{resolve}$ operator is the last case, where the abstract operator $\widehat{\mathbf{post}}$ and the abstract $\widehat{\mathbf{eval}}$ are used to calculate possible jump targets.

Note that the precision of the abstract domain influences the precision of the control flow analysis.

We are now ready to state constraints such that all solutions of these constraints are solutions to the control flow analysis. The first component is the Cartesian abstract domain $D : \mathbf{A} \to L$, which maps addresses to elements of the abstract domain. The idea is that D captures the data flow facts derived from the program. The second component is the set of edges $F \subseteq \mathbf{Edge}$ which stores the edges we produce by using the $\mathbf{resolve}$ operator. Finally, we provide initial abstract elements $\iota^a \in L$ for every location $a \in \mathbf{A}$. Then the constraints are as follows:

$$F \supseteq \bigcup_{a \in \mathbf{A}} \mathbf{resolve}_a(D(a)) \tag{1}$$

$$D(a) \sqsupseteq \bigsqcup_{(a', stmt, a) \in F} \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \sqcup \iota^a \tag{2}$$

Note how it pays off that edges are labeled. The partial control flow graph F does not only store the a priori unknown targets on the guarded jumps, but also the conditions (assume statements) which have to be satisfied to reach them. This information can be used by the abstract $\widehat{\mathbf{post}}$ to propagate precise information.

The system of constraints (1) and (2) corresponds to a function

$$\begin{aligned} G : \left((\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}\right) &\to \left((\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}\right) \\ G(D, F) &\mapsto (D', F'), \text{ where} \\ F' &= \bigcup_{a \in \mathbf{A}} \mathbf{resolve}_a(D(a)), \\ D'(a) &= \bigsqcup_{(a', stmt, a) \in F} \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \sqcup \iota^a. \end{aligned}$$

The connection between constraints (1) and (2) and control flow analysis is stated in the following theorem (detailed proof in Appendix A[1]), whereby correctness notably depends on $\iota^{\mathbf{start}} \in L$:

Theorem 1. *Given a program in the language* JUMP *and a trace* $\sigma = (s_i)_{0 \le i \le n}$*, such that* $s_0(pc) = \mathbf{start}$ *and* $s_0 \in \gamma(\iota^{\mathbf{start}})$*, every solution* (D, F) *of the constraints (1) and (2) satisfies* $s_n \in \gamma(D(s_n(pc)))$ *and* $TCFG(\sigma) \subseteq F$*.*

The proof is a straightforward induction on the length of traces using the properties we require from the abstract domain. We immediately obtain:

Corollary 1. *Given a program in the language* JUMP *and a solution* (D, F) *of the constraints (1) and (2), where* $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$*,* F *is a superset of the CFG.*

The Cartesian abstract domain $\mathbf{A} \to L$, equipped with pointwise ordering, i.e., $D_1 \sqsubseteq D_2 :\Leftrightarrow \forall a \in \mathbf{A}.\, D_1(a) \sqsubseteq D_2(a)$, is a complete lattice, because L is a complete

[1] Appendices included in the full version of this paper, available on `http://jakstab.org`

lattice. The power set $\mathbf{2^{Edge}}$ ordered by the subset relation $\subseteq$ is a complete lattice. The product lattice $(\mathbf{A} \to L) \times \mathbf{2^{Edge}}$, equipped with pointwise ordering, i.e., $(D_1, F_1) \sqsubseteq (D_2, F_2) :\Leftrightarrow D_1 \sqsubseteq D_2 \wedge F_1 \subseteq F_2$, is complete as both $\mathbf{A} \to L$ and $\mathbf{2^{Edge}}$ are complete. It can be easily seen that G is a monotone function on $(\mathbf{A} \to L) \times \mathbf{2^{Edge}}$. As $(\mathbf{A} \to L) \times \mathbf{2^{Edge}}$ is a complete lattice, we deduce from the Knaster-Tarski fixed point theorem [10] the existence of a least fixed point μ of the function G. Therefore, the following proposition immediately follows:

Proposition 1. *The combined control and data flow problem, i.e., the system of constraints (1) and (2), always has a unique best solution.*

3.4 Algorithms for Control Flow Reconstruction

For the purpose of algorithm design we will focus on abstract domains L satisfying the ascending chain condition (ACC). We now present two CFG-reconstruction algorithms. The first algorithm (Algorithm 1) is generic and gives an answer to the "chicken and egg" problem as it computes a sound overapproximation of the CFG by an intertwined control and data flow analysis. We stress the fact that the order in which the CFG reconstruction is done may only affect efficiency but not precision. The second algorithm (Algorithm 2) is an extension of the classical worklist algorithm and is geared towards practical implementation.

The generic algorithm maintains a Cartesian abstract domain $D : \mathbf{A} \to L$ and a partial control flow graph $F \subseteq \mathbf{Edge}$. $D(a)$ is initialized by $\iota^{\mathbf{start}}$ for $a = \mathbf{start}$ (line 3) and by $\bot$ for $a \neq \mathbf{start}$ (line 2). As we do not know anything about the control flow graph of the program yet, we start with F as the empty set (line 4). The algorithm iterates its main loop as long as it can find an unsatisfied inequality (line 7, 8). Thus the algorithm essentially searches for violations of constraints (1) and (2). If the generic algorithm finds at least one not yet satisfied inequality, it nondeterministically picks a single unsatisfied inequality and updates it (lines 9 to 14).

We now state the correctness of Algorithm 1 for abstract domains L that satisfy the ascending chain condition (detailed proof in Appendix B):

Theorem 2. *Given a program in the language* JUMP*, where* $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$*, the generic CFG-reconstruction algorithm (Algorithm 1) computes a sound overapproximation of the CFG and terminates in finite time. Furthermore it returns the most precise result with respect to the precision of the abstract domain L regardless of the non-deterministic choices made in line 9.*

Proof (sketch). The algorithm terminates because $(\mathbf{A} \to L) \times \mathbf{2^{Edge'}}$, where $\mathbf{Edge'}$ is the finite subset of $\mathbf{Edge}$ that consists of all the edges that are potentially part of the program, satisfies the ascending chain condition. The fact that the algorithm always computes the most precise result heavily depends on the existence of the unique least fixed point μ of G. It is easy to show that the generic algorithm computes this least fixed point μ. As the least fixed point is the best possible result with respect to the precision of the abstract domain, it is always the most precise regardless of the non-deterministic choices made in line 9.

Input: a JUMP-program, its set of addresses $\mathbf{A}$ including $\mathbf{start}$, and the abstract domain $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$ together with an initial value $\iota^{\mathbf{start}}$
Output: a control flow graph

```
1  begin
2      forall a ∈ A \ {start} do D(a) := ⊥;
3      D(start) := ι^start;
4      F := ∅;
5      while true do
6          Choices := ∅;
7          if ∃(a', stmt, a) ∈ F. post^[[stmt]](D(a')) ⋢ D(a) then Choices := Choices ∪ {do_p};
8          if ∃a ∈ A. resolve_a(D(a)) ⊈ F then Choices := Choices ∪ {do_r};
9          if ∃u ∈ Choices choose u ∈ U    /* non-deterministic choice */
10             switch u do
11                 case do_p choose (a', stmt, a) ∈ F where
                   post^[[stmt]](D(a')) ⋢ D(a)
12                     D(a) := post^[[stmt]](D(a')) ⊔ D(a);
13                 case do_r choose a ∈ A where resolve_a(D(a)) ⊈ F
14                     F := resolve_a(D(a)) ∪ F;
15         else
16             return F;
17 end
```

Algorithm 1. Generic CFG-reconstruction Algorithm

The worklist algorithm (Algorithm 2) is a specific strategy for executing the generic algorithm, where the partial control flow graph $F \subseteq \mathbf{Edge}$ is not kept as a variable, but implicit in the abstract values of the program locations. The initialization of D (lines 2, 3) is the same as in the generic algorithm. The algorithm maintains a worklist W, where it stores the edges for which data flow facts should be propagated later on. Every time the algorithm updates the information $D(a)$ at a location a (lines 3, 8), it calls the $\mathbf{resolve}$ operator (lines 4, 9) to calculate the edges which should be added to W. In every iteration of the main loop (lines 5 to 9) the algorithm non-deterministically picks an edge from the worklist by calling choose (line 6), and then shortens the worklist by calling rest (line 6). Subsequently, it checks for the received edge $(a', stmt, a)$, if an update is necessary (line 7), and in the case it is, it proceeds as already described.

From the correctness of the generic algorithm (1) we obtain the correctness of the worklist algorithm (proof in Appendix C):

Corollary 2. *Given a program in our language* JUMP*, where* $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$*, the worklist CFG-reconstruction algorithm (Algorithm 2) computes a sound overapproximation of the CFG and terminates in finite time. Furthermore it returns the most precise result with respect to the precision of the abstract domain* L *regardless of the non-deterministic choices made in line 6.*

Input: a JUMP-program, its set of addresses $\mathbf{A}$ including $\mathbf{start}$, and the abstract domain $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$ together with an initial value $\iota^{\mathbf{start}}$
Output: a control flow graph

```
1  begin
2      forall a ∈ A \ {start} do D(a) := ⊥;
3      D(start) := ι^start;
4      W := resolve_start(D(start));
5      while W ≠ ∅ do
6          ((a', stmt, a), W) := (choose(W), rest(W));
7          if post^[[stmt]](D(a')) ⋢ D(a) then
8              D(a) := post^[[stmt]](D(a')) ⊔ D(a);
9              W := add(W, resolve_a(D(a)));
10     F := ∅;
11     forall a ∈ A do
12         F := F ∪ resolve_a(D(a));
13     return F;
14 end
```

Algorithm 2. Worklist CFG-Reconstruction Algorithm

Proof (sketch). The worklist terminates because L satisfies the ascending chain condition. As the generic algorithm can always simulate the updates made by the worklist algorithm, the result computed by the worklist algorithm is always less or equal to the result of the generic algorithm, which is the least fixed point of G. On the other hand it can be shown that if the algorithm terminates, the result is greater or equal to the least fixed point of G.

Note that if the abstract domain L does not satisfy the ascending chain condition, it is possible to enhance the algorithms by using a widening operator to guarantee termination of the analysis. Such an algorithm would achieve a valid overapproximation of the CFG but lose the best approximation result stated in the above theorems, due to the imprecision induced by widening.

4 Instantiation of the Framework in the JAKSTAB Tool

We implemented the worklist algorithm for control flow reconstruction (Algorithm 2) in our disassembly and static analysis tool JAKSTAB [6]. JAKSTAB works on X86 executables, and translates them into an intermediate language that is similar in style but more complex than JUMP. We designed an abstract domain supporting constant propagation through registers (globally) and indirect memory locations (local to basic blocks) to parameterize the analysis, which yielded better results than the most widely used commercial disassembler IDA Pro. In this section we demonstrate how JAKSTAB integrates with our framework and sketch its abstract domain.

For supporting memory constants, JAKSTAB has to maintain an abstract representation of the store. When only dealing with memory accesses through constant addresses

	x	y	$m[x+2]$
start: $x := x + 2$	$(x + 2)$	$\top$	$\top$
2: $m[x] := 5$	$(x + 2)$	$\top$	5
3: $x := x + 1$	$(x + 3)$	$\top$	5
4: $x := x + 3$	$(x + 6)$	$\top$	5
5: $y := m[x - 4]$	$(x + 6)$	5	5
6: halt	$(x + 6)$	5	5

Fig. 2. Simple example for constant propagation through symbolic store locations. Abstract values calculated by the analysis are shown on the right.

(which is the case for global variables), this is trivial, since the store then behaves just like additional variables/registers. In compiled code, however, local variables are laid out on the stack, relative to the top of the current stack frame. They are manipulated by indirect addressing through the stack base pointer. For example, the instruction `mov [ebp - 4], 0` assigns 0 to the local variable at the top of the current stack frame. The exact value of the stack pointer, however, is only determined at runtime. Therefore, to successfully propagate constants through stack variables, our analysis must be able to handle indirect memory accesses symbolically, i.e., use symbolic store mappings from expressions to arbitrary expressions. The same holds true for fields in dynamically allocated memory structures, whose addresses are not statically known, either.

Support for symbolic store locations requires symbolic constants. Consider the simple program in Figure 2. The value of x is non-constant (because it is part of the input) and thus initialized to $\top$. To still propagate the assignment of 5 to the store location pointed to by x from line 2 to 5, the value of x has to be propagated symbolically, by forward substituting and partially evaluating the expressions. To this end, the lattice of abstract variable values contains symbolic expressions as an additional level of abstraction between integers and $\top$. Consequently, the mapping from store indices to integers has to be extended to a mapping $\mathbf{Exp} \to \mathbf{Exp}$. The join $\sqcup$ of the lattice for two elements with distinct values of the program counter is implemented by removing all symbolic mappings, retaining only mappings of variables to integers and from integer store locations to integers. This causes the scope of symbolic constant propagation to be limited to a single basic block. It also has the effect that the lattice L, which is of infinite height and satisfies the ascending chain condition; join points in loops always cause removal of mappings, thus every abstract state can only hold a finite number of mappings. Since ascending chains in the lattice remove one mapping per height level, the chains will always reach $\top$ after a finite number of steps.

The use of symbolic values has other implications as well. For updating symbolic values, the abstract $\widehat{\mathbf{post}}$ uses a substitution function that substitutes variables and memory expressions recursively with symbolic values from the abstract state. For substituting memory values, an aliasing check of store indices has to be performed. The abstract evaluation function $\widehat{\mathbf{eval}}$, which is used by our framework to resolve branch targets, uses substitution of symbolic store locations as well but ignores resulting symbolic values and only returns either integers, $\top$, or $\bot$. The concretization function γ maps each element of L to all concrete valuations matching the integer constants, disregarding symbolic mappings.

Using this abstract domain, JAKSTAB has already achieved good precision in reconstructing the control flow of executables [6]. Note that in the analysis of compiled applications, there are some cases when calls cannot be resolved by our current implementation. Most of the instances of unresolved control flow are due to function pointers inside structures being passed as parameters through several procedure calls. The local propagation of memory values in the abstract domain is currently not precise enough to capture such cases. Improvement of the propagation of memory values is a particular focus of ongoing work. The number of function pointer structures greatly depends on the implementation language of the program and the API functions used. In low level C code, the overwhelming majority of indirect calls result from compiler optimizations storing the addresses of frequently used API functions in registers, which JAKSTAB can reliably resolve.

5 Related Work

The problem of extracting a precise control flow graph from binaries has risen in several different communities of computer science. An obvious area is *reverse engineering* and in particular *decompilation*, where one aims to recover information about the original program, or preferably a close approximation of the original source code, from a compiled binary [11,12,13]. The compiler literature knows the concept of *link-time-* and *post-link-optimizers* [7,14], which exploit the fact that the whole program including libraries and hand-written assembly routines can be analyzed and optimized at link-time, i.e., after all code has been translated to binary with the symbol information still present. Precise tools for determining worst case execution time (WCET) of programs running on real time systems also have to process machine code, since they need to take compiler optimizations into account, and thus face similar problems of reconstructing the control flow [1,5,15]. Other applications of binary analysis include binary instrumentation [16], binary translation [17], or profiling [4].

Another prominent area of research that requires executable analysis is advanced *malware detection*. While classical malware detection relied on searching executables for binary strings (*signatures*) of known viruses, more recent advanced techniques focus on detecting patterns of malicious *behavior* by means of static analysis and model checking [18,19]. In this application domain, independence of the analysis from symbol information and compiler idioms is imperative, since malicious code is especially likely to have its symbols removed or to even be specially protected from analysis.

Due to the interest from these different communities, there has been a number of contributions to improving the results from disassembly. The literature contains a number of practical approaches to disassembly, which do not try to formulate a generalizable strategy. Schwarz et al. [2] describe a technique that uses an improved linear sweep disassembly algorithm, using relocation information to avoid misinterpreting data in a code segment. Subsequently, they run a recursive traversal algorithm on each function and compare results, but no attempt is made to recover from mismatching disassembly results. Harris and Miller [4] rely on identifying compiler idioms to detect procedures in the binary and to resolve indirect jumps introduced by jump tables. Cifuentes and van Emmerik [20] present a method to analyze jump tables by backward slicing through

register assignments and computing compound target expressions for the indirect jumps. These compound expressions are then matched against three compiler-specific patterns of implementing switch statements.

There have also been several proposals for more general frameworks for reconstructing the control flow from binaries. In Section 2.2, we already discussed the approach by De Sutter et al. [7], which targets code with symbol and relocation information and uses an overapproximating unknown-node for unresolved branch targets. In his bottom-up disassembly strategy, Theiling [1] assumes architectures in which all jump targets can be computed directly form the instruction, effectively disallowing indirect jumps. For extending his method to indirect jumps, he also suggests the use of an overapproximating unknown node.

Kästner and Wilhelm [5] describe a top-down strategy for structuring executables into procedures and basic blocks. For this to work, they require that code areas of procedures must not overlap, that there must be no data between or inside procedures, and that explicit labels for all possible targets of indirect jumps are present. Compilers, however, commonly generate procedures with overlapping entry and exit points, even if the control flow graphs of the procedures are completely separate, so their top-down structuring approach cannot be used in general without specific assumptions about the compiler or target architecture.

The advanced executable analysis tool Codesurfer/X86, presented by Balakrishnan and Reps [3], extracts information about relations between values and computes an approximation of possible values based on the abstract domain of value sets. For disassembly, they rely on the capabilities of the commercial disassembler IDA Pro. While they are able to resolve missing control flow edges through value set analysis, their separation from disassembly prevents that newly discovered code locations can be disassembled. Furthermore, CodeSurfer/X86 is vulnerable to errors introduced by the heuristics based disassembly strategy of IDA Pro.

Although operating at higher language levels, the decompilation approach of Chang et al. [13] is similar in spirit to our framework. They connect abstract interpreters operating at different language levels, executing them simultaneously. One can interpret our data flow analysis and control flow reconstruction as separate decompilation stages of their framework. However, we do not restrict the execution order but allow nondeterministic fixed point iteration over both analyses and are still able to prove that the resulting control flow graph is optimal.

6 Conclusion

We have built a solid foundation for the concept of disassembling binary code by defining a generic abstract interpretation framework for control flow reconstruction. While analysis of machine code often requires ad hoc solutions and has many pitfalls, we believe that it greatly helps in the design of disassemblers and binary analysis tools to know that data flow guided disassembly does not suffer from a “chicken and egg” problem. Based on our framework, we plan to further extend our own disassembler JAKSTAB with an improved abstract domain to further reduce the need for overapproximation of control flow.

References

1. Theiling, H.: Extracting safe and precise control flow from binaries. In: 7th Int'l. Workshop on Real-Time Computing and Applications Symp (RTCSA 2000), pp. 23–30. IEEE Computer Society, Los Alamitos (2000)
2. Schwarz, B., Debray, S.K., Andrews, G.R.: Disassembly of executable code revisited. In: 9th Working Conf. Reverse Engineering (WCRE 2002), pp. 45–54. IEEE Computer Society, Los Alamitos (2002)
3. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
4. Harris, L.C., Miller, B.P.: Practical analysis of stripped binary code. SIGARCH Comput. Archit. News 33(5), 63–68 (2005)
5. Kästner, D., Wilhelm, S.: Generic control flow reconstruction from assembly code. In: 2002 Jt. Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES 2002-SCOPES 2002), pp. 46–55. ACM Press, New York (2002)
6. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 423–427. Springer, Heidelberg (2008)
7. De Sutter, B., De Bus, B., De Bosschere, K.: Link-time binary rewriting techniques for program compaction. ACM Trans. Program. Lang. Syst. 27(5), 882–945 (2005)
8. Chang, P.P., Mahlke, S.A., Chen, W.Y., Hwu, W.W.: Profile-guided automatic inline expansion for C programs. Softw., Pract. Exper. 22(5), 349–369 (1992)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
10. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5(2), 285–309 (1955)
11. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Softw., Pract. Exper. 25(7), 811–829 (1995)
12. van Emmerik, M., Waddington, T.: Using a decompiler for real-world source recovery. In: 11th Working Conf. Reverse Engineering (WCRE 2004), pp. 27–36. IEEE Computer Society Press, Los Alamitos (2004)
13. Chang, B., Harren, M., Necula, G.: Analysis of low-level code using cooperating decompilers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 318–335. Springer, Heidelberg (2006)
14. Schwarz, B., Debray, S.K., Andrews, G.R.: PLTO: A link-time optimizer for the intel IA-32 architecture. In: Proc. Workshop on Binary Translation, WBT 2001 (2001)
15. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
16. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proc. ACM SIGPLAN 2007 Conf. Programming Language Design and Implementation (PLDI 2007), pp. 89–100. ACM Press, New York (2007)
17. Cifuentes, C., van Emmerik, M.: UQBT: Adaptive binary translation at low cost. IEEE Computer 33(3), 60–66 (2000)
18. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
19. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symp. Security and Privacy (S&P 2005), pp. 32–46. IEEE Computer Society, Los Alamitos (2005)
20. Cifuentes, C., van Emmerik, M.: Recovery of jump table case statements from binary code. Sci. Comput. Program. 40(2-3), 171–188 (2001)

SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities

Vincent Laviron[1] and Francesco Logozzo[2]

[1] École Normale Supérieure, 45, rue d'Ulm, Paris, France
Vincent.Laviron@ens.fr
[2] Microsoft Research, Redmond, WA, USA
logozzo@microsoft.com

Abstract. We introduce Subpolyhedra (SubPoly) a new numerical abstract domain to infer and propagate linear inequalities. SubPoly is as expressive as Polyhedra, but it drops some of the deductive power to achieve scalability. SubPoly is based on the insight that the reduced product of linear equalities and intervals produces powerful yet scalable analyses. Precision can be recovered using hints. Hints can be automatically generated or provided by the user in the form of annotations.

We implemented SubPoly on the top of `Clousot`, a generic abstract interpreter for `.Net`. `Clousot` with SubPoly analyzes very large and complex code bases in few minutes. SubPoly can efficiently capture linear inequalities among hundreds of variables, a result well-beyond state-of-the-art implementations of Polyhedra.

1 Introduction

The goal of an abstract interpretation-based static analyzer is to statically infer properties of the execution of a program that can be used to check its specification. The specification usually includes the absence of runtime exceptions (division by zero, integer overflow, array index out of bounds ...) and programmer annotations in the form of preconditions, postconditions, object invariants and assertions ("contracts" [23]). Proving that a piece of code satisfies its specification often requires discovering numerical invariants on program variables.

The concept of abstract domain is central in the design and the implementation of a static analyzer [9]. Abstract domains capture the properties of interest on programs. In particular *numerical* abstract domains are used to infer numerical relationships among program variables. Cousot and Halbwachs introduced the Polyhedra numerical abstract domain (Poly) in [11]. Poly infers all the linear inequalities on the program variables. The application and scalability of Poly has been severely limited by its performance which is worst-case exponential (easily attained in practice). To overcome this shortcoming and to achieve scalability, new numerical abstract domains have been designed either considering only inequalities of a particular shape (weakly relational domains) or fixing *ahead* of the analysis the maximum number of linear inequalities to be considered (bounded domains). The first class includes Octagons (which capture properties in the form

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 229–244, 2009.

```
class StringBuilder {
  int ChunkLen; char[] ChunkChars;

  public void Append(int wb, int count) {
    Contract.Requires(wb >= 2 * count);
    if (count + ChunkLen > ChunkChars.Length)
(*)   CopyChars(wb, ChunkChars.Length - ChunkLen);
    // ... }

  private void CopyChars(int wb, int len) {
    Contract.Requires(wb >= 2 * len);
  // ...  }
```

Fig. 1. An example extracted from `mscorlib.dll`. `Contract.Requires(...)` expresses method preconditions. Proving the precondition of `CopyChars` requires propagating an invariant involving three variables and non-unary coefficients.

$\pm \mathtt{x} \pm \mathtt{y} \leq c$) [24], TVPI ($a \cdot \mathtt{x} + b \cdot \mathtt{y} \leq c$) [29], Pentagons ($\mathtt{x} \leq \mathtt{y} \wedge a \leq \mathtt{x} \leq b$) [22], Stripes ($\mathtt{x} + a \cdot (\mathtt{y} + \mathtt{z}) > b$) [14] and Octahedra ($\pm \mathtt{x}_0 \cdots \pm \mathtt{x}_n \leq c$) [7]. The latter includes constraint template matrices (which capture at most m linear inequalities) [28] and methods to generate polynomial invariants *e.g.* [25,26].

Although impressive results have been achieved using weakly relational and bounded abstract domains, we experienced situations where the full *expressive* power of Poly is required. As an example, let us consider the code snippet of Fig. 1, extracted from `mscorlib.dll`, the main library of the `.Net` framework. Checking the precondition at the call site ($*$) involves (i) *propagating* the constraints $\mathtt{wb} \geq 2 \cdot \mathtt{count}$ and $\mathtt{count} + \mathtt{ChunkLen} > \mathtt{ChunkChars.Length}$; and (ii) *deducing* that $\mathtt{wb} \geq 2 \cdot (\mathtt{ChunkChars.Length} - \mathtt{ChunkLen})$. The aforementioned weakly relational domains cannot be used to check the precondition: Octahedra do not capture the first constraint (it involves a constraint with a non-unary coefficient); TVPI do not propagate the second constraint (it involves three variables); Pentagons and Octagons cannot represent any of the two constraints; Stripes can propagate both constraints, but because of the incomplete closure it cannot deduce the precondition. Bounded domains do the job, provided we fix *before* the analysis the template of the constraints. This is inadequate for our purposes: The analysis of a *single* method in `mscorlib.dll` may involve hundreds of constraints, whose shape cannot be fixed ahead of the analysis, *e.g.* by a textual inspection. Poly easily propagates the constraints. However, in the general case the price to pay for using Poly is too high: the analysis will be limited to few dozens of variables.

Subpolyhedra. We propose a new abstract domain, Subpolyhedra (SubPoly), which has the same *expressive* power as Poly, but it drops some inference power to achieve scalability: SubPoly exactly represents and propagates linear inequalities containing hundreds of variables and constraints. SubPoly is based on the fundamental insight that the reduced product of linear equalities, LinEq [17], and intervals, Intv [9], can produce very powerful yet efficient program analyses. SubPoly can represent linear inequalities using slack variables, *e.g.* $\mathtt{wb} \geq 2 \cdot \mathtt{count}$

is represented in SubPoly by $\mathtt{wb} - 2 \cdot \mathtt{count} = \beta \wedge \beta \in [0, +\infty]$. As a consequence, SubPoly easily proves that the precondition for `CopyChars` is satisfied at the call site $(*)$. In general the join of SubPoly is less precise than the one on Poly, so that it may not infer *all* the linear inequalities. Hints, either automatically generated or provided by the user, help recover precision.

Cardinal operations for SubPoly are: (i) the reduction, which propagates the information between LinEq and Intv; (ii) the join, which derives a compact yet precise upper approximation of two incoming abstract states; and (iii) the hint generator, which recovers information lost at join points.

```
void Foo(int i, int j) {
  int x = i, y = j;
  if (x <= 0) return;
  while (x > 0) { x--; y--; }
  if (y == 0) Assert(i == j); }
```

Fig. 2. An example from [27]. SubPoly infers the loop invariant $\mathtt{x} - \mathtt{i} = \mathtt{y} - \mathtt{j} \wedge \mathtt{x} \geq 0$, propagates it and proves the assertion.

Reduction. Let us consider the example in Fig. 2, taken from [27]. The program contains operations and predicates that can be exactly represented with Octagons. Proving that the assertion is not violated requires discovering the loop invariant $\mathtt{x} - \mathtt{y} = \mathtt{i} - \mathtt{j} \wedge \mathtt{x} \geq 0$. The loop invariant cannot be fully represented in Octagons: it involves a relation on four variables. Bounded numerical domains are unlikely to help here as there is no way to syntactically figure out the required template. The LinEq component of SubPoly infers the relation $\mathtt{x} - \mathtt{y} = \mathtt{i} - \mathtt{j}$. The Intv component of SubPoly infers the loop invariant $\mathtt{x} \in [0, +\infty]$, which in conjunction with the negation of the guard implies that $\mathtt{x} \in [0, 0]$. The reduction of SubPoly propagates the interval, refining the linear constraint to $\mathtt{y} = \mathtt{j} - \mathtt{i}$. This is enough to prove the assertion (in conjunction with the if-statement guard). It is worth noting that unlike [27] SubPoly does not require any hypothesis on the order of variables to prove the assertion.

Join and Hints. Let us consider the code in Fig. 3, taken from [15]. The loop invariant required to prove that the assertion is unreached (and hence that the program is correct) is $\mathtt{x} \leq \mathtt{y} \leq 100 \cdot \mathtt{x} \wedge \mathtt{z} = 10 \cdot \mathtt{w}$. Without hints, SubPoly can only infer $\mathtt{z} = 10 \cdot \mathtt{w}$. *Template* hints, inspired by [28], are used to recover linear inequalities that are dropped by the imprecision of the join: In the example the template is $\mathtt{x} - \mathtt{y} \leq b$, and the analysis automatically figures out that $b = 0$. *Planar Convex hull* hints, inspired by [29], are used to introduce at join points linear inequalities derived by a planar convex hull: In the example it helps the analysis figure out that $\mathtt{y} \leq 100 \cdot \mathtt{x}$. It is worth noting that SubPoly does not need any of the techniques of [15] to infer the loop invariant.

2 Abstract Interpretation Background

We assume the concrete domain to be the complete Boolean lattice of environments, *i.e.* $\mathsf{C} = \langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$, where $\Sigma = [\mathtt{Vars} \to \mathbb{Z}]$. An abstract domain A is a tuple $\langle \mathsf{D}, \gamma, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \triangledown, \rho \rangle$. The set of abstract elements D is related to the concrete domain by a *monotonic* concretization function

```
int x = 0, y = 0, w = 0, z = 0;
while (...) {
  if (...) { x++; y += 100; }
  else if (...) { if (x >= 4) { x++; y++; } }
  else if (y > 10 * w && z >= 100 * x) { y = -y; }

  w++; z += 10; }
if (x >= 4 && y <= 2) Assert(false);
```

Fig. 3. An example from [15]. SubPoly infers the loop invariant $\mathtt{x} \leq \mathtt{y} \leq 100{\cdot}\mathtt{x} \wedge \mathtt{z} = 10{\cdot}\mathtt{w}$, propagates it out of the loop, and proves that the assertion is unreached.

$\gamma \in [\mathsf{D} \to \mathsf{C}]$. With an abuse of notation, we will not distinguish between an abstract domain and the set of its elements. The approximation order $\sqsubseteq$ soundly approximates the concrete order: $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_1 \implies \gamma(\mathsf{d}_0) \subseteq \gamma(\mathsf{d}_1)$. The smallest element is $\bot$, the largest element is $\top$. The join operator $\sqcup$ satisfies $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_0 \sqcup \mathsf{d}_1 \wedge \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \sqcup \mathsf{d}_1$. The meet operator $\sqcap$ satisfies $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqcap \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \wedge \mathsf{d}_0 \sqcap \mathsf{d}_1 \sqsubseteq \mathsf{d}_1$. The widening $\triangledown$ ensures the convergence of the fixpoint iterations, *i.e.* it satisfies: (i) $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_0 \triangledown \mathsf{d}_1 \wedge \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \triangledown \mathsf{d}_1$; and (ii) for each sequence of abstract elements $\mathsf{d}_0, \mathsf{d}_1, \ldots \mathsf{d}_k$ the sequence defined by $\mathsf{d}_0^\triangledown = \mathsf{d}_0, \mathsf{d}_1^\triangledown = \mathsf{d}_0^\triangledown \triangledown \mathsf{d}_1 \ldots \mathsf{d}_k^\triangledown = \mathsf{d}_{k-1}^\triangledown \triangledown \mathsf{d}_k$ is ultimately stationary. In general, we do not require abstract elements to be in some canonical or closed form, *i.e.* there may exist $\mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}$, such that $\mathsf{d}_0 \neq \mathsf{d}_1$, but $\gamma(\mathsf{d}_0) = \gamma(\mathsf{d}_1)$. The *reduction* operator $\rho \in [\mathsf{D} \to \mathsf{D}]$ puts an abstract element into a (pseudo-)canonical form without adding or losing any information: $\forall \mathsf{d}.\ \gamma(\rho(\mathsf{d})) = \gamma(\mathsf{d}) \wedge \rho(\mathsf{d}) \sqsubseteq \mathsf{d}$. We do not require ρ to be idempotent. New abstract domains can be systematically derived by cartesian composition or functional lifting [10]. Following [8], we use the dot-notation to denote point wise extensions.

Intervals. The abstract domain of interval environments is $\langle \mathsf{Intv}, \gamma_{\mathsf{Intv}}, \sqsubseteq_{\mathsf{Intv}}, \bot_{\mathsf{Intv}}, \top_{\mathsf{Intv}}, \sqcup_{\mathsf{Intv}}, \sqcap_{\mathsf{Intv}}, \triangledown_{\mathsf{Intv}} \rangle$. The abstract elements are maps from program variables to open intervals. The concretization of an interval environment i is $\gamma_{\mathsf{Intv}}(\mathsf{i}) = \{s \in \Sigma \mid \forall \mathtt{x} \in \mathrm{dom}(\mathsf{i}).\ \mathsf{i}(\mathtt{x}) = [a, b] \wedge a \leq s(\mathtt{x}) \leq b\}$. The order is interval inclusion, the bottom element is the empty interval, the top is the interval $[-\infty, +\infty]$, the join is the smallest interval which contains the two arguments, the meet is interval intersection, and the widening keeps the stable bounds. The reduction is the identity function. All the domain operations can be implemented to take linear time.

Linear Equalities. The abstract domain of linear *equalities* is $\langle \mathsf{LinEq}, \gamma_{\mathsf{LinEq}}, \sqsubseteq_{\mathsf{LinEq}}, \bot_{\mathsf{LinEq}}, \top_{\mathsf{LinEq}}, \sqcup_{\mathsf{LinEq}}, \sqcap_{\mathsf{LinEq}} \rangle$. The elements are sets of linear equalities, their meaning is given by the set of concrete states which satisfy the constraints, *i.e.* $\gamma_{\mathsf{LinEq}} = \lambda \mathsf{I}.\ \{s \in \Sigma \mid \forall (\sum a_i \cdot \mathtt{x}_i = b) \in \mathsf{I}.\ \sum a_i \cdot s(\mathtt{x}_i) = b\}$. The order is sub-space inclusion, the bottom is the empty space, the top is the whole space, the join is the smallest space which contains the two arguments, the meet is space intersection. LinEq has finite height, so the join suffices to ensure analysis termination. The reduction is Gaussian elimination. The complexity of

the domain operations is subsumed by the complexity of Gaussian elimination, which is cubic.

Polyhedra. The abstract domain of linear *inequalities* is $\langle \mathsf{Poly}, \gamma_{\mathsf{Poly}}, \sqsubseteq_{\mathsf{Poly}}, \bot_{\mathsf{Poly}}, \top_{\mathsf{Poly}}, \sqcup_{\mathsf{Poly}}, \sqcap_{\mathsf{Poly}}, \triangledown_{\mathsf{Poly}} \rangle$. The elements are sets of linear inequalities, the concretization is the set of concrete states which satisfy the constraints *i.e.* $\gamma_{\mathsf{Poly}} = \lambda \mathsf{p}.\ \{s \in \Sigma \mid \forall (\sum a_i \cdot \mathtt{x}_i \leq b) \in \mathsf{p}.\ \sum a_i \cdot s(\mathtt{x}_i) \leq b\}$, the order is the polyhedron inclusion, the bottom is the empty polyhedron, the top is the whole space, the join is the convex hull, the meet is just the union of the set of constraints, and the widening preserves the inequalities stable among two successive iterations. The reduction infers the set of generators and removes the redundant inequalities. The cost of the Poly operations is subsumed by the cost of the conversion between the algebraic representation (set of inequalities) and the geometric representation (set of generators) used in the implementation [1]. In fact, some operations require the algebraic representation (*e.g.* $\sqcap_{\mathsf{Poly}}$), some require the geometrical representation (*e.g.* $\sqcup_{\mathsf{Poly}}$), and some others require both (*e.g.* $\sqsubseteq_{\mathsf{Poly}}$). The conversion between the two representations is exponential in the number of variables, and it cannot be done better [18].

3 Subpolyhedra

We introduce the numerical abstract domain of Subpolyhedra, SubPoly. The main idea of SubPoly is to combine Intv and LinEq to capture complex linear *inequalities*. Slack variables are introduced to replace inequality constraints with equalities.

Variables. A variable $\mathtt{v} \in \mathtt{Vars}$ can either be a *program* variable ($\mathtt{x} \in \mathtt{Var_P}$) or a *slack* variable ($\beta \in \mathtt{Var_S}$). A slack variable β has associated information, denoted by $\mathtt{info}(\beta)$, which is a linear form $a_1 \cdot \mathtt{v}_1 + \cdots + a_k \cdot \mathtt{v}_k$. Let $\kappa \equiv \sum a_i \cdot \mathtt{x}_i + \sum b_j \cdot \beta_j = c$ be a linear equality: $\mathtt{s}_\kappa = \sum_{\mathtt{x}_i \in \mathtt{Var_P}} a_i \cdot \mathtt{x}_i$ denotes the partial sum of the monomials involving just program variables; $\mathtt{Var_P}(\kappa) = \{\mathtt{x}_i \mid a_i \cdot \mathtt{x}_i \in \kappa, a_i \neq 0\}$ and $\mathtt{Var_S}(\kappa) = \{\beta_j \mid b_j \cdot \beta_j \in \kappa, b_j \neq 0\}$ denote respectively the program variables and the slack variables in κ. The generalization to inequalities and sets of equalities and inequalities is straightforward.

Elements. The elements of SubPoly belong to the reduced product LinEq $\otimes$ Intv [10]. Inequalities are represented in SubPoly with slack variables: $\sum a_i \cdot \mathtt{x}_i \leq c \iff \sum a_i \cdot \mathtt{x}_i - c = \beta \wedge \beta \in [-\infty, 0]$ (β is a fresh slack variable with the associated information $\mathtt{info}(\beta) = \sum a_i \cdot \mathtt{x}_i$).

Concretization. A subpolyhedron can be interpreted as a polyhedron by projecting out the slack variables: $\gamma_S^{\mathsf{Poly}} \in [\mathsf{SubPoly} \to \mathsf{Poly}]$ is $\gamma_S^{\mathsf{Poly}} = \lambda \langle \mathsf{I};\ \mathsf{i} \rangle.\ \pi_{\mathtt{Var_S}}(\mathsf{I} \cup \{a \leq \mathtt{v} \leq b \mid \mathsf{i}(\mathtt{v}) = [a, b]\})$, where π denotes the projection of variables in Poly. The concretization $\gamma_S \in [\mathsf{SubPoly} \to \mathcal{P}(\Sigma)]$ is then $\gamma_S = \gamma_{\mathsf{Poly}} \circ \gamma_S^{\mathsf{Poly}}$.

Approximation Order. The order on SubPoly may be defined in terms of order over Poly. Given two subpolyhedra $\mathsf{s}_0, \mathsf{s}_1$, the most precise order relation $\sqsubseteq_S^*$ is $\mathsf{s}_0 \sqsubseteq_S^* \mathsf{s}_1 \iff \gamma_S^{\mathsf{Poly}}(\mathsf{s}_0) \sqsubseteq_{\mathsf{Poly}} \gamma_S^{\mathsf{Poly}}(\mathsf{s}_1)$. However, $\sqsubseteq_S^*$ may be too expensive to compute: it involves mapping subpolyhedra in the dual representation of Poly.

```
if(...)                          if(...)
 { assume x - y <= 0; }           { assume x == y; assume y <= z; }
else                             else
 { assume x - y <= 5; }           { assume x <= y; assume y == z; }
```

(a) (b)

Fig. 4. Examples illustrating the need for Step 1 in the join algorithm

This can easily cause an exponential blow up. We define a weaker approximation order relation which first tries to find a renaming θ for the slack variables, and then checks the pairwise order. Formally ($\cdot \xrightarrow{\text{inj}} \cdot$ denotes an injective function):

$$\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle \sqsubseteq_S \langle \mathsf{l}_1;\ \mathsf{i}_1\rangle \iff \exists\theta.\ \mathtt{Var_S}(\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle) \xrightarrow{\text{inj}} \mathtt{Var_S}(\langle \mathsf{l}_1;\ \mathsf{i}_1\rangle).$$
$$\forall \beta \in \mathtt{Var_S}(\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle).\ \mathtt{info}(\beta) = \mathtt{info}(\theta(\beta)) \wedge \theta(\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle) \dot{\sqsubseteq} \langle \mathsf{l}_1;\ \mathsf{i}_1\rangle.$$

In general $\sqsubseteq_S \subsetneq \sqsubseteq_S^*$. In practice, $\sqsubseteq_S$ is used to check if a fixpoint has been reached. A weaker order relation means that the analysis may perform some extra widening steps, which may introduce precision loss. However, we found the definition of $\sqsubseteq_S$ satisfactory in our experience.

Bottom. A subpolyhedron is equivalent to bottom if after a reduction one of the two components is bottom: $\mathsf{s} = \bot_S \iff \rho(\mathsf{s}) = \langle \mathsf{l}, \mathsf{i}\rangle \wedge (\mathsf{i} = \dot{\bot}_{\mathsf{Intv}} \vee \mathsf{l} = \bot_{\mathsf{LinEq}})$.

Top. A subpolyhedron is top if both components are top: $\mathsf{s} = \top_S \iff \mathsf{s} = \langle \mathsf{l}, \mathsf{i}\rangle \wedge \mathsf{i} = \top_{\mathsf{Intv}} \wedge \mathsf{l} = \top_{\mathsf{LinEq}}$.

Linear form Evaluation. Let s be a linear form: $[\![\mathsf{s}]\!] \in [\mathsf{SubPoly} \to \mathsf{Intv}]$ denotes the evaluation of s in a subpolyhedron after the reduction has inferred the tightest bounds: $[\![\sum a_i \cdot \mathsf{v}_i]\!]\langle \mathsf{l};\ \mathsf{i}\rangle = \textbf{let}\ \langle \mathsf{l}^*;\ \mathsf{i}^*\rangle = \rho(\langle \mathsf{l};\ \mathsf{i}\rangle)\ \textbf{in} \sum a_i \cdot \mathsf{i}^*(\mathsf{v}_i)$.

Join. The join $\sqcup_S$ is computed in three steps. First, inject the information of the slack variables into the abstract elements. Second, perform the pairwise join on the saturated arguments. Third, add the constraints that are implied by the two operands of the join, but that were not preserved by the previous step. The join, parameterized by the reduction ρ, is defined by the Algorithm 1 (We let $\underline{0} = 1, \underline{1} = 0$). We illustrate the join with examples. We postpone the discussion of the reduction to Sect. 4.

Example 1 (Steps 1 & 2). Let us consider the code in Fig. 4(a). After the assumption, the abstract states on the true branch and the false branch are respectively: $\mathsf{s}_0 = \langle \mathtt{x} - \mathtt{y} = \beta_0;\ \beta_0 \in [-\infty, 0]\rangle$ and $\mathsf{s}_1 = \langle \mathtt{x} - \mathtt{y} = \beta_1;\ \beta_1 \in [-\infty, 5]\rangle$. The information associated with the slack variables is $\mathtt{info}(\beta_0) = \mathtt{info}(\beta_1) = \mathtt{x} - \mathtt{y}$. At the join point we apply Algorithm 1. Step 1 refines the abstract states by introducing the information associated with the slack variables: $\mathsf{s}'_0 = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 0]\rangle$ and $\mathsf{s}'_1 = \langle \mathtt{x} - \mathtt{y} = \beta_1 = \beta_0;\ \beta_1 \in [-\infty, 5]\rangle$. Step 2 requires the reduction of the operands. The interval for β_1 (resp. β_0) in s'_0 (resp. s'_1) is refined: $\rho(\mathsf{s}'_0) = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 0], \beta_1 \in [-\infty, 0]\rangle$ and $\rho(\mathsf{s}'_1) = \langle \mathtt{x} - \mathtt{y} = \beta_1 = \beta_0;\ \beta_0 \in [-\infty, 5], \beta_1 \in [-\infty, 5]\rangle$. The pairwise join gets the expected invariant: $\mathsf{s}_\sqcup = \rho(\mathsf{s}'_0) \dot{\sqcup} \rho(\mathsf{s}'_1) = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 5], \beta_1 \in [-\infty, 5]\rangle$. □

Algorithm 1. The join $\sqcup_S$ on Subpolyhedra

input $\langle \mathsf{l}_i;\ \mathsf{i}_i\rangle \in \mathsf{SubPoly}$, $i \in \{0,1\}$

let $\langle \mathsf{l}'_i;\ \mathsf{i}'_i\rangle = \langle \mathsf{l}_i;\ \mathsf{i}_i\rangle$

{Step 1. Propagate the information of the slack variables}
for all $\beta \in \mathtt{Var_S}(\mathsf{l}_i) \setminus \mathtt{Var_S}(\mathsf{l}_{\underline{i}})$ **do**
  $\langle \mathsf{l}'_{\underline{i}};\ \mathsf{i}'_{\underline{i}}\rangle := \langle \mathsf{l}'_{\underline{i}} \sqcap_{\mathsf{LinEq}} \{\beta = \mathtt{info}(\beta)\};\ \mathsf{i}'_{\underline{i}}\rangle$

{Step 2. Perform the point-wise join on the saturated operands}
let $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup\rangle = \rho(\langle \mathsf{l}'_0;\ \mathsf{i}'_0\rangle) \dot{\sqcup} \rho(\langle \mathsf{l}'_1;\ \mathsf{i}'_1\rangle)$

{Step 3. Recover the lost information }
let D_i be the linear equalities dropped from l'_i at the previous step
for all $\kappa \in D_i$ **do**
  let $\mathsf{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!]\langle \mathsf{l}'_{\underline{i}};\ \mathsf{i}'_{\underline{i}}\rangle$
  if κ contains no slack variable **then**
    if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      let β be a fresh slack variable
      $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup\rangle := \langle \mathsf{l}_\sqcup \sqcap_{\mathsf{LinEq}} \{\beta = \kappa\};\ \mathsf{i}_\sqcup \sqcap_{\mathsf{Intv}} \{\beta = \mathsf{i}_{s_\kappa} \sqcup_{\mathsf{Intv}} [0,0]\}\rangle$
  else if κ contains exactly one slack variable β **then**
    if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup\rangle := \langle \mathsf{l}_\sqcup \sqcap_{\mathsf{LinEq}} \{\kappa\}; \mathsf{i}_\sqcup \sqcap_{\mathsf{Intv}} \{\beta = \mathsf{i}_{s_\kappa} \sqcup_{\mathsf{Intv}} \mathsf{i}_i(\beta)\}\rangle$
return $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup\rangle$

Example 2 (Non-trivial information for slack variables). Let us consider the code snippet in Fig. 4(b). The abstract states to be joined are $\langle \mathtt{x}-\mathtt{y}=0, \mathtt{y}-\mathtt{z}=\beta_0; \beta_0 \in [-\infty,0]\rangle$ and $\langle \mathtt{y}-\mathtt{z}=0, \mathtt{x}-\mathtt{y}=\beta_1; \beta_1 \in [-\infty,0]\rangle$. The associated information are $\mathtt{info}(\beta_0) = \mathtt{y}-\mathtt{z}$ and $\mathtt{info}(\beta_1) = \mathtt{x}-\mathtt{y}$. Step 1 allows to refine the abstract states with the slack variable information, and hence to infer that after the join $\mathtt{x} \leq \mathtt{y}$ and $\mathtt{y} \leq \mathtt{z}$. □

The two examples above show the importance of introducing the information associated with slack variables in Step 1 and the reduction in Step 2. Without those, the relation between the slack variables and the program point where they were introduced would have been lost.

The join of LinEq is *precise* in that if a linear equality is implied by both operands, then it is implied by the result too. The same for the join of Intv. The pairwise join in $\mathsf{LinEq} \otimes \mathsf{Intv}$ may drop some inequalities. Some of those can be recovered by the refinement step. The next example illustrates it.

Example 3 (Step 3). Let us consider the code in Fig. 5(a). The analysis of the two branches of the conditional produces the abstract states: $\mathtt{s}_0 = \langle \mathtt{x}-3\cdot\mathtt{y}=0;\ \top_{\mathsf{Intv}}\rangle$ and $\mathtt{s}_1 = \langle \mathtt{x}=0, \mathtt{y}=1;\ \mathtt{x} \in [0,0], \mathtt{y} \in [1,1]\rangle$. The reduction ρ does not refine the states (we already have the tightest bounds). The point-wise join produces the abstract state $\top_S$. Step 3 identifies the dropped constraints: $D_0 = \{\mathtt{x}-3\cdot\mathtt{y}=0\}$ and $D_1 = \{\mathtt{x}=0, \mathtt{y}=1\}$. The algorithm inspects them to check if they are satisfied by the "other" branch. The constraint in D_0 is also satisfied in the false branch: $[\![\mathtt{x}-3\cdot\mathtt{y}]\!](\mathtt{s}_1) = [-3,-3]$ $(\neq \top_{\mathsf{Intv}})$. Therefore it can be safely added to

Algorithm 2. The widening ∇_S on Subpolyhedra

input $\langle \mathsf{l}_i;\ \mathsf{i}_i \rangle \in \mathsf{SubPoly},\ i \in \{0,1\}$

let $\langle \mathsf{l}'_i;\ \mathsf{i}'_i \rangle = \langle \mathsf{l}_i;\ \mathsf{i}_i \rangle$

{Step 1. Propagate the information of the slack variables}
for all $\beta \in \mathtt{Var_S}(\mathsf{l}_0) \setminus \mathtt{Var_S}(\mathsf{l}_1)$ **do**
  $\langle \mathsf{l}'_0;\ \mathsf{i}'_0 \rangle := \langle \mathsf{l}'_0 \sqcap_{\mathsf{LinEq}} \{\beta = \mathtt{info}(\beta)\};\ \mathsf{i}'_0 \rangle$

{Step 2. Perform the point-wise widening}
let $\langle \mathsf{l}_\nabla;\ \mathsf{i}_\nabla \rangle = \langle \mathsf{l}'_0;\ \mathsf{i}'_0 \rangle \dot{\nabla} \rho(\langle \mathsf{l}'_1;\ \mathsf{i}'_1 \rangle)$

{Step 3. Recover the lost information }
let D_0 be the linear equalities dropped from l'_0 at the previous step
for all $\kappa \in D_0$ **do**
  let $\mathsf{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!] \langle \mathsf{l}'_1;\ \mathsf{i}'_1 \rangle$
  if κ contains no slack variables **then**
    if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      let β be a fresh slack variable
      $\langle \mathsf{l}_\nabla;\ \mathsf{i}_\nabla \rangle := \langle \mathsf{l}_\nabla \sqcap_{\mathsf{LinEq}} \{\beta = \kappa\};\ \mathsf{i}_\nabla \sqcap_{\mathsf{Intv}} \{\beta = [0,0] \nabla \mathsf{i}_{s_\kappa}\} \rangle$
  else if κ contains exactly one slack variable β **then**
    if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      $\langle \mathsf{l}_\nabla;\ \mathsf{i}_\nabla \rangle := \langle \mathsf{l}_\nabla \sqcap_{\mathsf{LinEq}} \{\kappa\};\ \mathsf{i}_\nabla \sqcap_{\mathsf{Intv}} \{\beta = \mathsf{i}_0(\mathtt{v}) \nabla \mathsf{i}_{s_\kappa}\} \rangle$

return $\langle \mathsf{l}_\nabla;\ \mathsf{i}_\nabla \rangle$

the result. The constraints of D_2 do not hold on the left branch and they are discarded. The abstract state after the join is $\mathsf{s}_\sqcup = \langle \mathtt{x} - 3 \cdot \mathtt{y} = \beta;\ \beta \in [-3, 0] \rangle$. □

Meet. The meet $\sqcap_S$ is simply the pairwise meet on $\mathsf{LinEq} \otimes \mathsf{Intv}$.

Widening. The definition of the widening (Algorithm 2) is similar to the join, with the main differences that: (i) the information associated to slack variables is propagated only in one direction; (ii) only the right argument is saturated; and (iii) the recovery step is applied only to one of the operands. Those hypotheses avoid the well-known problems of interaction between reduction, refinement and convergence of the iterations [24].

Example 4 (Refinement step for the widening). Let us consider the code snippet in Fig. 5(b). The entry state to the loop is $\mathsf{s}_0 = \langle \mathtt{i} - \mathtt{k} = 0;\ \top_{\mathsf{Intv}} \rangle$. The state after one iteration is $\mathsf{s}_1 = \langle \mathtt{i} - \mathtt{k} = 1;\ \top_{\mathsf{Intv}} \rangle$. We apply the widening operator. Step 1 does not refine the states as there are no slack variables. The pairwise widening of Step 2 loses all the information. Step 3 recovers the constraint $\mathtt{k} \leq \mathtt{i}$:

```
if(...) { assume x == 3 * y; }
else    { x = 0; y = 1; }
```
(a)

```
i := k;
while(...) i++;
assert i >= k;
```
(b)

Fig. 5. Examples illustrating the need for the Step 3 in the join and the widening

$D_0 = \{\mathtt{i} - \mathtt{k} = 0\}$ contains no slack variables and $[\![\mathtt{i} - \mathtt{k}]\!](\mathtt{s}_1) = [1, 1]$ so that $\mathtt{s}_\nabla = \langle \mathtt{i} - \mathtt{k} = \beta;\ \beta \in [0, +\infty] \rangle$. □

Theorem 1 (Fixpoint convergence). *The operator defined in Algorithm 2 is a widening. Moreover, $\sqsubseteq_S$ can be used to check that the fixpoint iterations eventually stabilize.*

4 Reduction for Subpolyhedra

The reduction in SubPoly infers tighter bounds on linear forms and hence on program variables. Reduction is cardinal to fine tuning the precision/cost ratio. We propose two reduction algorithms, one based on linear programming, ρ_{LP}, and the other on basis exploration, ρ_{BE}. Both of them have been implemented in `Clousot`, our abstract interpretation-based static analyzer for `.Net` [4].

Linear Programming-Based Reduction. A linear programming problem is the problem of maximizing (or minimizing) a linear function subject to a finite number of linear constraints. We consider *upper bounding* linear problems (UBLP) [6], *i.e.* problems in the form (n is the number of variables, m is the number of equations):

$$\begin{array}{ll} \text{maximize} & c \cdot \mathtt{v}_k \qquad\qquad k \in 1 \ldots n, c \in \{-1, +1\} \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} a_{ij} \cdot \mathtt{v}_j = b_j \quad (i = 1, \ldots m) \quad \text{and} \quad l_j \leq \mathtt{v}_j \leq u_j \quad (j = 1, \ldots n). \end{array}$$

The Linear programming-based reduction ρ_{LP} is trivially an instance of UBLP: To infer the tightest upper bound (resp. lower bound) on a variable $\mathtt{v}_k$ in a subpolyhedron $\langle \mathsf{l};\ \mathsf{i} \rangle$ instantiate UBLP with $c = 1$ (resp. $c = -1$) subject to the linear equalities l and the numerical bounds i. UBLP can be solved in polynomial time [6]. However, polynomial time algorithms for UBLP do not perform well in practice. The Simplex method [12], exponential in the worst-case, in practice performs a lot better than other known linear programming algorithms [30]. The Simplex algorithm works by visiting the *feasible bases* (informally, the vertexes) of the polyhedron associated with the constraints. At each step, the algorithm visits the adjacent basis (vertex) that maximizes the current value of the objective by the largest amount. The iteration strategy of the Simplex guarantees the convergence to a basis which exhibits the optimal value for the objective.

The advantages of using Simplex for ρ_{LP} are that: (i) it is well-studied and optimized; (ii) it is complete in $\mathbb{R}$, *i.e.* it finds the best solution over real numbers; and (iii) it guarantees that all the information is propagated at once: $\rho_{LP} \circ \rho_{LP} = \rho_{LP}$.

The drawbacks of using Simplex are that (i) the computation over machine floating point may introduce imprecision or unsoundness in the result; and (ii) the reduction ρ_{LP} requires to solve $2 \cdot n$ UBLP problems to find the lower bound and the upper bound for each of the n variables in an abstract state. We have observed (i) in our experiences (cf. Sect. 6). There exist methods to circumvent the problem at the price of extra computational cost, *e.g.* using arbitrary

Algorithm 3. The reduction algorithm ρ_{BE}, parametrized by the oracle δ

input $\langle \mathsf{l};\ \mathsf{i} \rangle \in \mathsf{SubPoly}$, $\delta \in \mathcal{P}(\{\zeta \mid \zeta \text{ is a basis change}\})$

Put l into row echelon form. Call the result l'
let $\langle \mathsf{l}^*, \mathsf{i}^* \rangle = \langle \mathsf{l}', \mathsf{i} \rangle$
for all $\zeta \in \delta$ **do**
  $\mathsf{l}^* := \zeta(\mathsf{l}^*)$
  for all $\mathtt{v}_k + a_{k+1} \cdot \mathtt{v}_{k+1} + \cdots + a_n \cdot \mathtt{v}_n = b \in \mathsf{l}^*$ **do**
    $\mathsf{i}^* := \mathsf{i}^*[\mathtt{v}_k \mapsto \mathsf{i}^*(\mathtt{v}_k) \sqcap_{\mathsf{Intv}} [\![b - a_{k+1} \cdot \mathtt{v}_{k+1} + \cdots + a_n \cdot \mathtt{v}_n]\!](\mathsf{i}^*)]$
return $\langle \mathsf{l}^*, \mathsf{i}^* \rangle$

precision rationals, or a combination of machine floating arithmetic and precise arithmetic. Even if (i) is solved, we observed that (ii) dominates the cost of the reduction, in particular in the presence of abstract states with a large number of variables: the $2 \cdot n$ UBLP problems are *disjoints* and there is no easy way to share the sequence of bases visited by the Simplex algorithm over the different runs of the algorithm for the same abstract state.

Basis Exploration-Based Reduction. We have developed a new reduction ρ_{BE}, less subject to the drawbacks from floating point computation than ρ_{LP}, which enables a better tuning of the precision/cost ratio than the Simplex. The basic ideas are: (i) to fix *ahead* of time the bases we want to explore; and (ii) to refine at each step the variable bounds. The reduction ρ_{BE}, parametrized by a set of changes of basis δ, is formalized by Algorithm 3. First, we put the initial set of linear constraints into triangular form (row echelon form). Then, we apply the basis changes in δ and we refine all the variables *in the basis*. With respect to ρ_{LP}, ρ_{BE} is faster: (i) the number of bases to explore is statically bounded; (ii) at each step, k variables may be refined at once.

In theory, ρ_{BE} is an abstraction of ρ_{LP}, in that it may not infer the *optimal* bounds on variables (it depends on the choice of δ). In practice, we found that ρ_{LP} is much more numerically stable and it can infer better bounds than ρ_{LP}. The reason is in the handling of numerical errors in the computation. Suppose we are seeking a (lower or upper) bound for a variable using the Simplex. If we detect a numerical error (*i.e.*, a huge coefficient in the exact arithmetic computation), the only sound solution is to stop the iterations, and return the current value of the objective function as the result. On the other hand, when we detect a numerical error in ρ_{BE}, we can just skip the current basis (abstraction), and move to the next one in δ.

We are left with the problem of defining δ. We have two instantiations for it: a linear explorer and combinatorial explorer. The algorithm in Sect. 9.3.3 of [13] may also be used when the all the variables are known to be positive.

Linear Explorer (δ_L). The linear bases explorer is based on the empirical observation that in most cases having some variable $\mathtt{v}_0$ in the basis and some other variable $\mathtt{v}_1$ out of the basis is enough to infer good bounds. The explorer generates a sequence of bases δ_L with the property that for each unordered pair of

distinct variables $\langle \mathtt{v}_0, \mathtt{v}_1 \rangle$, it exists $\zeta \in \delta_L$ such that $\mathtt{v}_0$ is in the basis and $\mathtt{v}_1$ is not. The sequence δ_L is defined as $\delta_L = \{\zeta_i \mid i \in [0, n],\ \mathtt{v}_i \dots \mathtt{v}_{(i+m-1) \bmod n} \text{ are in basis for } \zeta_i\}$.

Example 5. (Reduction with the linear explorer) Let the initial state be $\mathtt{s} = \langle \mathtt{v}_0 + \mathtt{v}_2 + \mathtt{v}_3 = 1, \mathtt{v}_1 + \mathtt{v}_2 - \mathtt{v}_3 = 0;\ \mathtt{v}_0 \in [0,2], \mathtt{v}_1 \in [0,3] \rangle$, so that $\delta_L = \{\{\mathtt{v}_0, \mathtt{v}_1\}, \{\mathtt{v}_1, \mathtt{v}_2\}, \{\mathtt{v}_2, \mathtt{v}_3\}, \{\mathtt{v}_3, \mathtt{v}_0\}\}$. The reduction $\rho_{BE}(\mathtt{s})$ contains the tightest bounds for $\mathtt{v}_2, \mathtt{v}_3$: $\langle \mathtt{v}_2 + \frac{1}{2} \cdot \mathtt{v}_0 + \frac{1}{2} \cdot \mathtt{v}_1 = 0, \mathtt{v}_3 + \frac{1}{2} \cdot \mathtt{v}_0 - \frac{1}{2} \cdot \mathtt{v}_1 = 0; \mathtt{v}_0 \in [0,2], \mathtt{v}_1 \in [0,3], \mathtt{v}_2 \in [0, \frac{5}{2}], \mathtt{v}_3 \in [-\frac{1}{2}, 1] \rangle$. □

Properties of δ_L are that: (i) each variable appears exactly m times in the basis; (ii) it can be implemented efficiently as the basis change from ζ_i to $\zeta_{i+1}, i \in [0, n-1]$ requires just one variable swap; (iii) in general it is not idempotent: it may be the case that $\rho_L \circ \rho_L \neq \rho_L$; (iv) the result may depend on the initial order of variables, as shown by the next example.

Example 6 (Incompleteness of the linear explorer). Let us consider an initial state $\mathtt{s} = \langle \mathtt{v}_0 + \mathtt{v}_1 + \mathtt{v}_2 = 0, \mathtt{v}_3 + \mathtt{v}_1 = 0;\ \mathtt{v}_2 \in [0,1], \mathtt{v}_3 \in [0,1] \rangle$. The reduced state $\rho_{BE}(\mathtt{s}) = \langle \mathtt{v}_3 + \mathtt{v}_1 = 0, \mathtt{v}_2 + \mathtt{v}_0 - \mathtt{v}_1 = 0; \mathtt{v}_1 \in [-1,0], \mathtt{v}_2 \in [0,1], \mathtt{v}_3 \in [0,1] \rangle$ does not contain the bound $\mathtt{v}_0 \in [-1,1]$. □

Combinatorial Explorer (δ_C). The combinatorial explorer δ_C systematically visits all the bases. It generates all possible combinations of $\mathtt{m}$ variables trying to minimize the number of swaps at each basis change. It is very costly, but it finds the best bounds for each variable: it visits all the bases, in particular the one where the optimum is reached. The main advantage with respect to the Simplex is a better tolerance to numerical errors. However it is largely impractical because of (i) the huge cost; and (ii) the negligible gain of precision w.r.t. the use of δ_L that it showed in our benchmark examples.

5 Hints

The inference power of SubPoly can be increased using *hints*. Hints are linear functionals associated with a subpolyhedron $\mathtt{s}$. They represent some linear inequality that *may* hold in $\mathtt{s}$, but that it is not explicitly represented by a slack variable, or that it is not been checked to hold in $\mathtt{s}$ yet.

Hints increase the precision of joins and widenings. Let $\mathtt{h}$ be an hint, let $\mathtt{s}_0$ and $\mathtt{s}_1$ two subpolyhedra, and let $\mathtt{b} = [\![\mathtt{h}]\!](\mathtt{s}_0) \sqcup_{\mathsf{Intv}} [\![\mathtt{h}]\!](\mathtt{s}_1)$. If $\mathtt{b} \neq \top_{\mathsf{Intv}}$, then $\mathtt{h} \in \mathtt{b}$ holds in both $\mathtt{s}_0$ and $\mathtt{s}_1$, so that the constraint can be safely added to $\mathtt{s}_0 \sqcup_S \mathtt{s}_1$. That helps recovering linear inequalities that may have been dropped by the Algorithm 1. The situation for widening is similar, with the main difference that the number of hints should be bounded, to ensure convergence. Hints can be automatically generated during the analysis or they can be provided by the user in the form of annotations. In our current implementation, we have three ways to generate hints, inspired by existing solutions in the literature: program text, templates and planar convex hull. They provide very powerful hints, but some of them may be expensive.

Program Text Hints. They introduce a new hint each time a guard or assume statement (user annotation) is encountered in the analysis. This way, properties that are obvious when looking at the syntax of the program will be proved. Also, every time a slack variable β is removed, $\texttt{info}(\beta)$ is added to the hints. This is useful in the realistic case when SubPoly is used in conjunction with a heap analysis which may introduce unwanted renamings.

Template Hints. They consider hints of fixed shape [28]. For instance, hints in the form $\mathtt{x}_0 - \mathtt{x}_1$ guarantee a precision at least as good as difference bounds matrices [24], provided that the reduction is complete.

Planar Convex Hull Hint. It materializes new hints by performing the planar convex hull of the subpolyhedra to join [29]. First, it projects the interval components on every two-dimensional plane (there are a quadratic number of such planes). Then it performs the convex hull of the resulting pair of rectangles (in constant time, since the number of vertexes is at most eight). The resulting new linear constraints are a sound approximation by construction. They can be safely added to the result of the join.

6 Experience

We have implemented SubPoly on top of `Clousot`, our modular abstract interpretation-based static analyzer for `.Net` [3]. A stand-alone version of the SubPoly library is available for download [19]. `Clousot` directly analyzes MSIL, a bytecode target for more than seventy compilers (including C#, Managed C++, VB.NET, F#). Prior to the numerical analysis `Clousot` performs a heap analysis and an expression recovery analysis [21]. `Clousot` performs *intra*-procedural analysis and it supports assume-guarantee reasoning via `Foxtrot` annotations [4]. Contracts are expressed directly in the language as method calls and are persisted to MSIL using the normal compilation process of the source language. Classes and methods may be annotated with class invariants, preconditions and postconditions. Preconditions are asserted at call sites and assumed at the method

Assembly	Methods	Bounds Checked	SubPoly with ρ_{LP} Valid	%	Time	SubPoly with ρ_{BE} Valid	%	Time	Max Vars
`mscorlib.dll`	18 084	17 181	14 432	84.00	73:48 (3)	14 466	84.20	23:19 (0)	373
`System.dll`	13 776	11 891	10 225	85.99	58:15 (2)	10 427	87.69	14:45 (0)	140
`System.Web.dll`	22 076	14 165	13 068	92.26	24:41 (0)	13 078	92.33	6:33 (0)	182
`System.Design.dll`	11 419	10 519	10 119	96.20	26:07 (0)	10 148	96.47	5:18 (0)	73
Average				89.00			89.51		

Fig. 6. The experimental results of checking array creation and accesses in representative `.Net` assemblies. SubPoly is instantiated with two reductions: ρ_{LP} and ρ_{BE}. Time is in minutes. The number of methods that reached the timeout (two minutes) is in parentheses. The last column reports the maximum number of variables simultaneously related by a SubPoly abstract state.

entry point. Postconditions are assumed at call sites and asserted at the method exit point. `Clousot` also checks the absence of specific errors, *e.g.* out of bounds array accesses, null dereferences, buffer overruns, and divisions by zero.

Figure 6 summarizes our experience in analyzing array creations and accesses in four libraries shipped with `.Net`. The test machine is an ordinary 2.4Ghz dual core machine, running Windows Vista. The assemblies are directly taken from the `%WINDIR%\Microsoft\Framework\v2.0.50727` directory of the PC. The analyzed assemblies do not contain contracts (We are actively working to annotate the `.Net` libraries). On average, we were able to validate almost 89.5% of the proof obligations. We manually inspected some of the warnings issued for `mscorlib.dll`. Most of them are due to lack of contracts, *e.g.* an array is accessed using a method parameter or the return value of some helper method. However, we also found real bugs (dead code and off-by-one). That is remarkable considering that `mscorlib.dll` has been tested *in extenso*. We also tried SubPoly on the examples of [11,27,15,16], proving all of them.

Reduction Algorithms. We run the tests using the Simplex-based and the Linear explorer-based reduction algorithms. We used the Simplex implementation shipped with the Microsoft Automatic Graph Layout tool, widely tested and optimized. The results in Fig. 6 show that ρ_{LP} is significantly slower than ρ_{BE}, and in particular the analysis of five methods was aborted as it reached the two minutes time-out. Larger time-outs did not help.

SubPoly with the reduction ρ_{LP} validates less accesses than ρ_{BE}. Two reasons for that. First, it is slower, so that the analysis of some methods is aborted and hence their proof obligations cannot be validated. Second, our implementation of the Simplex uses floating point arithmetic which induces some loss of precision. In particular we need to read back the result (a `double`) into an interval of `int`s containing it. In general this may cause a loss of precision and even worse unsoundness. We experienced both of them in our tests. For instance the 39 "missing" proof obligations in `System.Web.dll` and `System.Design.dll` (validated using ρ_{BE}, but not with ρ_{LP}) are due to floating point imprecision in the Simplex. We have considered replacing a floating point-based Simplex with one using exact rationals. However, the Simplex has the tendency to generate coefficients with large denominators. The code we analyze contains many large constants which cause the Simplex to produce enormous denominators.

SubPoly with ρ_{BE} instantiated with the linear bases explorer perform very well in practice: it is extremely fast and precise. Our implementation uses 64 bits Rationals. When an arithmetic overflow is detected, we abstract away the current computation, *e.g.*, by removing the suitable row in the matrix representation. On the negative side, the result may depend on the variables order. A "bad" variable order may cause ρ_{BE} not to infer bounds tight enough. One solution is to iterate the application of ρ_{BE} (it is not idempotent). Other solutions are: (i) to reduce the number of variables by simplifying a subpolyhedron (less bases to explore); (ii) to mark variables which can be safely kept in the basis at all times: In the best case, only one basis needs to be explored. In the general case, it still makes the reduction more precise because the bases explored are more likely to give bounds on the variables.

Max Variables. It is worth noting that even if `Clousot` performs an intra-procedural analysis, the methods we analyze may be very complex, and they may require tracking linear inequalities among many abstract locations. Abstract locations are produced by the heap analysis [20], and they abstract stack locations and heap locations. Figure 6 shows that it is not uncommon to have methods which require the abstract state to track more than 100 variables. One single method of `mscorlib.dll` required to track relations among 373 distinct variables. SubPoly handles it: the analysis with ρ_{BE} took a little bit more than a minute. To the best of our knowledge those performances in presence of so many variables are largely beyond current Poly implementations. For instance, in some preliminary study we tried to instantiate `Clousot` with the Poly library included in Boogie [2]. The results were quite disappointing: under the same experimental conditions (except for a 5 minutes time out), the analysis of `System.dll` took 257 minutes, and the time out was reached more than 20 times. We did not notice any remarkable gain of precision using Poly. Furthermore, Poly is concerned by floating points soundness issues, too [5].

7 Conclusions

We introduced SubPoly, a new numerical abstract domain based on the combination of linear equalities and intervals. SubPoly can track linear inequalities involving hundreds of variables. We defined the operations of the abstract domain (order, join, meet, widening) and two reduction operators (one based on linear programming and another based on basis exploration). We found Simplex-based reduction quite unsatisfactory for program analysis purposes: because of floating point errors the result may be too imprecise or worse, unsound. We introduced then the basis exploration-based reduction, in practice more precise and faster.

SubPoly precisely propagates linear inequalities, but it may fail to infer some of them at join points. Precision can be recovered using hints either provided by the programmer in the form of program annotations; or automatically generated (at some extra cost). SubPoly worked fine on some well known examples in literature that required the use of Poly. We tried SubPoly on shipped code, and we showed that it scales to several hundreds of variables, a result far beyond the capabilities of existing Poly implementations.

Acknowledgments. Thanks to L. Nachmanson for providing us the Simplex implementation. Thanks to M. Fähndrich, J. Feret, S. Gulwani, C. Popeea and J. Smans for the useful discussions.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Sci. Comput. Program. 72(1) (2008)

2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for Object-Oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., Fähndrich, M., Logozzo, F.: Managed contract tools, `http://research.microsoft.com/downloads`
4. Barnett, M., Fähndrich, M.A., Logozzo, F.: Foxtrot and Clousot: Language agnostic dynamic and static contract checking for. Net. Technical Report MSR-TR-2008-105, Microsoft Research (2008)
5. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: APLAS 2008 (2008)
6. Chvátal, V.: Linear Programming. W.H. Freeman, New York (1983)
7. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 312–327. Springer, Heidelberg (2004)
8. Cousot, P.: The calculational design of a generic abstract interpreter. In: Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
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 (1977)
10. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979 (1979)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978 (1978)
12. Dantzig, G.B.: Programming in linear structures. Technical report, USAF (1948)
13. Feret, J.: Analysis of mobile systems by abstract interpretation. PhD thesis
14. Ferrara, P., Logozzo, F., Fähndrich, M.A.: Safer unsafe code in. Net. In: OOPSLA 2008 (2008)
15. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
16. Gulwani, S., Mehra, K., Chilimbi, T.: Speed: Precise and efficient static estimation of program computational complexity. In: POPL 2009 (2009)
17. Karr, M.: On affine relationships among variables of a program. Acta Informatica 6(2), 133–151 (1976)
18. Khachiyan, L., Boros, E., Borys, K., Elbassioni, K.M., Gurvich, M.: Generating all vertices of a polyhedron is hard. In: SODA 2006 (2006)
19. Laviron, V., Logozzo, F.: The Subpoly Library, `http://research.microsoft.com/downloads`
20. Logozzo, F.: Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 283–298. Springer, Heidelberg (2007)
21. Logozzo, F., Fähndrich, M.A.: On the relative completeness of bytecode analysis versus source code analysis. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 197–212. Springer, Heidelberg (2008)
22. Logozzo, F., Fähndrich, M.A.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: SAC 2008 (2008)
23. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Professional Technical Reference. Prentice-Hall, Englewood Cliffs (1997)
24. Miné, A.: The octagon abstract domain. In: WCRE 2001 (2001)
25. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL 2004 (2004)

26. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. 64(1) (2007)
27. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis using symbolic ranges. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007)
28. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
29. Simon, A., King, A., Howe, J.: Two variables per linear inequality as an abstract domain. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664. Springer, Heidelberg (2003)
30. Spielman, D.A., Teng, S.-H.: Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. J. ACM. 51(3) (2004)

Deciding Extensions of the Theories of Vectors and Bags

Patrick Maier

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh, Scotland
Patrick.Maier@ed.ac.uk

Abstract. Vectors and bags are basic collection data structures, which are used frequently in programs and specifications. Reasoning about these data structures is supported by established algorithms for deciding ground satisfiability in the theories of arrays (for vectors) and multisets (for bags), respectively. Yet, these decision procedures are only able to reason about vectors and bags in isolation, not about their combination.

This paper presents a decision procedure for the combination of the theories of vectors and bags, even when extended with a function bagof bridging between vectors and bags. The function bagof converts vectors into the bags of their elements, thus admitting vector/bag comparisons. Moreover, for certain syntactically restricted classes of ground formulae decidability is retained if the theory of vectors is extended further with a map function which applies uninterpreted functions to all elements of a vector.

1 Introduction

Vectors and bags are basic collection data structures, which are used frequently in programs and specifications. Reasoning about these data structures is supported by decision procedures for deciding the satisfiability of quantifier-free formulae in the theories of arrays (for vectors) and multisets (for bags), respectively. However, known decision procedures are essentially only able to reason about vectors and bags in isolation, whereas practical software verification problems often require non-trivial combinations.

Let us illustrate this problem with an example. Figure 1 shows a Java method `sendBulk` taking a message text `msg`, a group of recipients `group` (represented as an array of phone numbers) and a *resource manager* `mgr` holding (symbolic representations of) the resources required to send text messages to the recipients. As the cost of sending text messages may vary depending on the recipient, the state of a resource manager cannot be simply the number of messages that may be sent; instead it should be a multiset of resources, representing exactly how many messages may be sent to whom. In order to enforce the resource limit, at least at run-time, actual use of resources must be preceded by a call to the resource manager's `use` method, which checks whether the required resource is present and if so, deduces it, otherwise aborts the program. This is what's happening in the body of method `sendBulk`, which iterates over `group`, sending `msg` to each member by calling `SMS.send`, but only after checking for and using up the associated resource by calling `mgr.use`. This approach to run-time monitoring of resources via explicit resource managers has been described in [1], for example.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 245–259, 2009.

```
void sendBulk(String msg, PhoneNum[] group, ResourceMgr mgr) {
    for (int i=0; i < group.length; i++) {
        mgr.use(MessageResource(group[i]));
        SMS.send(msg, group[i]);
    }
}
```

$$\begin{aligned}
\mathit{PreCond} &\equiv \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group})) \subseteq \texttt{mgr} \\
\mathit{PostCond} &\equiv \backslash\texttt{old}(\texttt{mgr}) = \texttt{mgr} \uplus \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group})) \\
\mathit{LoopInv} &\equiv 0 \leq \texttt{i} \leq \texttt{group.length} \wedge \\
&\quad \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group[i:group.length]})) \subseteq \texttt{mgr} \wedge \\
&\quad \backslash\texttt{old}(\texttt{mgr}) = \texttt{mgr} \uplus \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group[0:i]})) \\
\mathit{VC} &\equiv \mathit{LoopInv} \wedge \neg \mathit{LoopInv}[\texttt{i}+1/\texttt{i}, \texttt{mgr}'/\texttt{mgr}] \wedge \texttt{i} < \texttt{group.length} \wedge \\
&\quad \mathrm{count}(\texttt{mgr}, \texttt{MessageResource(group[i])}) > 0 \wedge \\
&\quad \texttt{mgr} = \texttt{mgr}' \uplus [\![\texttt{MessageResource(group[i])}]\!]^{(1)}
\end{aligned}$$

Fig. 1. Java bulk messaging example: code and specification of send loop

Run-time monitoring provides dynamic guarantees of *resource safety*, as abuse of resources will be trapped. However, aborting a program midway is not always a desirable solution; it would be better if we could guarantee statically that a program will never even *attempt* to abuse resources. This is done in [2], which presents a type system for proving static resource safety in a programming language with explicit resource managers. When proving resource safety of a method like `sendBulk`, whether it is done via a type system as in [2] or in the more traditional way by generating verification conditions, the hard part is reasoning about constraints between the program variables. Ideally, we'd like to have fully automated theorem provers for this task.

Let us take a look at the constraints required to express invariants and pre- and postconditions for `sendBulk`, see the bottom half of Figure 1. Informally, the precondition states that `mgr` is a super-multiset of the vector `group`, when the latter is viewed as a multiset of resources. To express this view, we first need to convert `group` into a vector of resources (by applying the map function) and then into a multiset of resources (by applying the bagof function). The postcondition states that the old `mgr` splits into two multisets: the new `mgr` and the multiset of resources corresponding to the vector `group`. The loop invariant essentially combines pre- and postcondition, but for different slices of the vector `group`. The first conjunct bounds the loop variable `i`, the second is the precondition for the remainder of the loop, i. e., for the subvector from index `i` to the end, and the third is the effect of the loop so far, i. e., the postcondition for the subvector from index 0 up to (but excluding) `i`. The (negated) verification condition conjoins the loop invariant before, the negated loop invariant after the execution of the loop (arising by substituting the variables `i` and `mgr`), the loop condition, and the precondition (`mgr` has some resources corresponding to number `group[i]`) and effect (`mgr`$'$ holds one unit of resource less than `mgr`) of the loop body. Hence, to

verify the loop invariant of an example even this simple we must prove unsatisfiability of constraints about bags, vectors, subvectors, the map function for transforming vectors pointwise, and the bagof function for transforming vectors into multisets.

Decision procedures for vectors (or arrays) exist for quite some time; early work goes back to the late 1970s [6, 10]. Recently, [4] and [3] found expressive yet decidable extensions of the theory arrays by injectivity predicates and by restricted quantification over array indices, respectively. Decision procedures for bags (or multisets) have been published recently in [12] and [7, 8], where the latter supports a cardinality operator. However, decision procedures combining vectors and bags and linking them via the bagof function (or something similar) do not exist.

The main contribution of this paper is a decision procedure for ground satisfiability in the combination of the theories of vectors and bags extended with the function bagof. For certain syntactically restricted classes of ground formulae decidability is retained if the theory of vectors is extended further with a map_f function for transforming vectors pointwise by applying the uninterpreted function f. The decision procedure reduces formulae containing $\mathrm{bagof}(\cdot)$ to formulae without by instantiating universally quantified variables in the axiomatisation of the bagof function, eventually reducing the problem to the theories of vectors and bags. It relies on a decision procedure for the Array Property Fragment described in [3] and on a decision procedure for multisets with cardinality described in [7, 8].

Plan. Section 2 introduces some basic notation. Section 3 presents the theories of bags, vectors, map and bagof functions. Section 4 utilises known results to construct a decision procedure for the combination of the theories of bags and vectors (including map). Section 5 presents our main result: an extension of the decision procedure (and its proof of correctness) to cope with bagof.

2 Preliminaries

We work in the framework of *many-sorted first-order logic with equality*, assuming familiarity with the basic syntactic and semantic concepts. Below we fix some notation.

Throughout the paper, we fix three countably infinite and pairwise disjoint universes: a set $\mathcal{S}$ of *sorts*, a set $\mathcal{F}$ of *function symbols* and a set $\mathcal{X}$ of *variable symbols*. By S^+ we denote the set of non-empty words over a set S.

Signatures. A *decorated variable* x_s is a pair consisting of a variable $x \in \mathcal{X}$ and a sort $s \in \mathcal{S}$. A *decorated function symbol* f_w is a pair consisting of a function symbol $f \in \mathcal{F}$ and an *arity* $w \in \mathcal{S}^+$. A decorated function symbol c_s of arity $s \in \mathcal{S}$ is called a *decorated constant*. For the sake of readability, we may write decorated constants and function symbols in the form $c : s$ and $f : s_1 \times \ldots \times s_n \to s_0$ instead of c_s and $f_{s_0 s_1 \ldots s_n}$, respectively. We may drop decorations entirely if they are clear from the context.

A *(many-sorted) signature* Σ is a pair $\Sigma = \langle S, F \rangle$ where $S \subseteq \mathcal{S}$ is a non-empty finite set of sorts and $F \subseteq \mathcal{F} \times S^+$ is a set of decorated function symbols. We may write Σ^{S} and Σ^{F} to refer to S and F, respectively. If Σ_1 and Σ_2 are signatures then the *union*

$\Sigma_1 \cup \Sigma_2 = \langle \Sigma_1^{\mathrm{S}} \cup \Sigma_2^{\mathrm{S}}, \Sigma_1^{\mathrm{F}} \cup \Sigma_2^{\mathrm{F}} \rangle$ and *intersection* $\Sigma_1 \cap \Sigma_2 = \langle \Sigma_1^{\mathrm{S}} \cap \Sigma_2^{\mathrm{S}}, \Sigma_1^{\mathrm{F}} \cap \Sigma_2^{\mathrm{F}} \rangle$ are signatures, too. Two signatures Σ_1 and Σ_2 are *disjoint* if $\Sigma_1^{\mathrm{F}} \cap \Sigma_2^{\mathrm{F}} = \emptyset$, i. e., disjoint signatures do not share decorated function symbols but may share sorts.

Union and intersection induce a lattice structure on signatures. We denote the induced partial order by $\supseteq$, where $\Sigma_2 \supseteq \Sigma_1$ (in words: Σ_2 *extends* Σ_1) if $\Sigma_2^{\mathrm{S}} \supseteq \Sigma_1^{\mathrm{S}}$ and $\Sigma_2^{\mathrm{F}} \supseteq \Sigma_1^{\mathrm{F}}$. The *constant expansion* of Σ, denoted by $\hat{\Sigma}$, is the greatest signature extending Σ such that $\hat{\Sigma}^{\mathrm{S}} = \Sigma^{\mathrm{S}}$ and all function symbols in $\hat{\Sigma}^{\mathrm{F}} \setminus \Sigma^{\mathrm{F}}$ are constants, i. e., $\hat{\Sigma}$ provides infinitely many constants per sort.

Terms and Formulae. Let Σ be a signature. *Σ-terms* are well-sorted terms constructed from decorated function symbols in Σ^{F} and decorated variables in $\mathcal{X} \times \Sigma^{\mathrm{S}}$. A *ground* Σ-term is a variable-free Σ-term. If Σ is clear from the context, we may drop the prefix and write "term" instead of "Σ-term". We may refer to terms of sort $s \in \Sigma^{\mathrm{S}}$ as *s-terms*.

A Σ-atom is an equality[1] $t = t'$, where t and t' are Σ-terms of the same sort. A Σ-literal is a Σ-atom $t = t'$ or its negation $\neg(t = t')$, often written as $t \neq t'$. If we want to stress that the sort of left- and right-hand sides of a Σ-atom (resp.-literal) is s, we may refer to the atom (resp. literal) as s-atom (resp. s-literal). Σ-formulae are formed from Σ-atoms by the usual connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) and quantifiers ($\forall$, $\exists$) of first-order logic, inducing the usual notion of bound and free variables. A *Σ-sentence* is a Σ-formula without free variables, and a *Σ-theory* is a set of Σ-sentences. Note that a Σ-theory $\mathcal{T}$ is also a Σ'-theory, for all Σ' extending Σ. A *ground* Σ-formula is a quantifier-free Σ-sentence.

Algebras and Satisfiability. Let $\Sigma = \langle S, F \rangle$ be a signature. A *Σ-algebra* $\mathcal{A}$ is a pair $\langle S^{\mathcal{A}}, F^{\mathcal{A}} \rangle$ where $S^{\mathcal{A}}$ is a S-indexed family of carrier sets and $F^{\mathcal{A}}$ is a F-indexed family of functions on the carrier sets. More formally, $S^{\mathcal{A}} = \{s^{\mathcal{A}} | s \in \Sigma^{\mathrm{S}}\}$ is a family of non-empty and pairwise disjoint sets $s^{\mathcal{A}}$, and $F^{\mathcal{A}} = \{f^{\mathcal{A}}_{s_0 s_1 \ldots s_n} | f_{s_0 s_1 \ldots s_n} \in F\}$ is a family of functions $f^{\mathcal{A}}_{s_0 s_1 \ldots s_n}$ from $s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}}$ to $s_0^{\mathcal{A}}$. We extend the interpretation of function symbols in a Σ-algebra $\mathcal{A}$ homomorphically to ground Σ-terms t in the usual way, denoting the resulting element of the algebra by $t^{\mathcal{A}}$. Note that for all Σ' extending Σ, a Σ'-algebra $\mathcal{A}$ can also be viewed as a Σ-algebra.

The truth of a Σ-sentence ϕ in a Σ-algebra $\mathcal{A}$, denoted by $\mathcal{A} \models \phi$, is defined in the usual way. $\mathcal{A}$ is a *model* of a Σ-theory $\mathcal{T}$, also denoted by $\mathcal{A} \models \mathcal{T}$, if $\mathcal{A} \models \phi$ for all $\phi \in \mathcal{T}$. Given a Σ-algebra $\mathcal{A}$, the theory $\mathcal{T}(\mathcal{A})$ is the greatest Σ-theory which has $\mathcal{A}$ as a model. Given a class Δ of Σ-algebras, $\mathcal{T}(\Delta) = \bigcap_{\mathcal{A} \in \Delta} \mathcal{T}(\mathcal{A})$ is the greatest Σ-theory which has all algebras $\mathcal{A} \in \Delta$ as models.

Let $\mathcal{T}$ be a Σ-theory. A Σ-algebra $\mathcal{A}$ is a $\mathcal{T}$-model if $\mathcal{A} \models \mathcal{T}$. A $\hat{\Sigma}$-sentence ϕ is *$\mathcal{T}$-satisfiable* if there is a $\mathcal{T}$-model $\mathcal{A}$ which is a model of ϕ; note that $\mathcal{A}$ must be a $\hat{\Sigma}$-algebra. Two $\hat{\Sigma}$-sentences ϕ and ψ are *$\mathcal{T}$-equisatisfiable* if both are $\mathcal{T}$-satisfiable or neither is.

Given a subset $S' \subseteq \Sigma^{\mathrm{S}}$ of sorts, a Σ-theory $\mathcal{T}$ is *stably infinite w. r. t. S'* if every $\mathcal{T}$-satisfiable ground $\hat{\Sigma}$-formula ϕ has a $\mathcal{T}$-model $\mathcal{A}$ such that $s^{\mathcal{A}}$ is infinite for all $s \in S'$. $\mathcal{T}$ is *stably infinite* if it is stably infinite w. r. t. the set of all sorts Σ^{S}.

[1] We consider equality the only predicate symbol of the logic. Other predicates can be encoded as functions with a non-trivial codomain.

3 Theories

We introduce the signatures and theories used throughout this paper, see also Figure 2.

Elements. $\mathcal{T}_{\mathrm{E}}$ is a given theory of elements (of vectors and bags). Its signature Σ_{E} is arbitrary but must be disjoint from all other signatures introduced in this section. The theory $\mathcal{T}_{\mathrm{E}}$ is arbitrary, too, but must be decidable and stably infinite so it can be coupled with the theory of multisets, see Section 4.1.

Presburger arithmetic. Σ_{INT} is the signature of Presburger arithmetic, with one sort, two constants and four binary function symbols (for addition, subtraction, minimum and maximum). We introduce the binary predicate symbols $\leq$ and $<$ as abbreviations; we may write $s \leq t$ instead of $\min(s,t) = s$ and $s < t$ instead of $s \leq t \wedge s \neq t$.

The theory $\mathcal{T}_{\mathrm{INT}}$ of Presburger arithmetic is defined as the set of all Σ_{INT}-sentences which are true in $\mathcal{A}_{\mathrm{INT}}$, the standard Σ_{INT}-algebra which interprets the sort INT as the integers and constants and function symbols by their usual meaning.

Multisets. The signature Σ_{BAG} of multisets (with cardinality) extends the signature of Presburger arithmetic with element sorts and multiset sorts BAG_s, one per element sort s. For each element sort, Σ_{BAG} extends Σ_{INT} with a constant $[\![]\!]$ for the empty multiset, a singleton constructor $[\![\cdot]\!]^{(\cdot)}$ (taking an element and its multiplicity), the usual binary operations $\cap$, $\cup$, $\uplus$ for intersection, union and sum, a destructor $\mathrm{count}(\cdot,\cdot)$ for counting the frequency of an element in a multiset, and a destructor $|\cdot|$ for measuring the cardinality (i. e., the number of elements, taking into account their multiplicities) of a multiset. We introduce the binary predicate symbol $\subseteq$ as an abbreviation; we may write $s \subseteq t$ instead of $s \cap t = s$.

Due to the cardinality function, the theory of multisets cannot be finitely axiomatised in our logic.[2] Therefore, the theory $\mathcal{T}_{\mathrm{BAG}}$ of multisets is defined as the set of all Σ_{BAG}-sentences that are true of Δ_{BAG}, the class of standard Σ_{BAG}-algebras. $\mathcal{A}$ is a standard Σ_{BAG}-algebra if it interprets the sort INT as the integers, the sorts BAG_s as the finite multisets over the interpretations of the sorts s, and the constants and function symbols by their usual meanings. Note that the theory $\mathcal{T}_{\mathrm{INT}}$ is contained in $\mathcal{T}_{\mathrm{BAG}}$; stable infiniteness will be relevant in Section 4.1.

Lemma 1. *$\mathcal{T}_{\mathrm{BAG}}$ is stably infinite.*

Vectors. We represent vectors by finite arrays of elements indexed by consecutive integers. The signature Σ_{VEC} of vectors extends the signature of Presburger arithmetic with element sorts and vector sorts VEC_s, one per element sort s. For each element sort, Σ_{VEC} extends Σ_{INT} with two destructors $\mathrm{fst}(\cdot)$ and $\mathrm{end}(\cdot)$ for accessing the first and last (more precisely, the first beyond the last) index of a vector, a destructor $\cdot[\cdot]$ for reading an element of a vector, a constructor $\mathrm{const}(\cdot,\cdot,\cdot)$ for creating a vector filled with a multiple occurrences of the same element, a constructor $\cdot[\cdot{:}\cdot]$ for slicing the subvector in between two indices out of a vector, and a constructor $\cdot\{\cdot \leftarrow \cdot\}$ for updating a vector at an index.

[2] See [7] for an axiomatisation in a first-order logic extended with an *infinite sum* quantifier.

Σ_{E}-theory $\mathcal{T}_{\mathrm{E}}$ of elements

Σ_{E} arbitrary signature disjoint from all signatures below,
$\mathcal{T}_{\mathrm{E}}$ arbitrary stably infinite theory with decidable ground $\mathcal{T}_{\mathrm{E}}$-satisfiability problem.

Σ_{INT}-theory $\mathcal{T}_{\mathrm{INT}}$ of Presburger arithmetic

$$\Sigma^{\mathrm{S}}_{\mathrm{INT}} = \{\mathrm{INT}\}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{INT}} = \big\{0, 1 : \mathrm{INT},\ +, -, \min, \max : \mathrm{INT} \times \mathrm{INT} \to \mathrm{INT}\big\}$$
$\mathcal{T}_{\mathrm{INT}} = \mathcal{T}(\mathcal{A}_{\mathrm{INT}})$ where $\mathcal{A}_{\mathrm{INT}}$ is the standard Σ_{INT}-algebra.

Σ_{BAG}-theory $\mathcal{T}_{\mathrm{BAG}}$ of multisets with cardinality

$$\Sigma^{\mathrm{S}}_{\mathrm{BAG}} = \Sigma^{\mathrm{S}}_{\mathrm{INT}} \cup \Sigma^{\mathrm{S}}_{\mathrm{E}} \cup \{\mathrm{BAG}_s \mid s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{BAG}} = \Sigma^{\mathrm{F}}_{\mathrm{INT}} \cup \big\{|\cdot| : \mathrm{BAG}_s \to \mathrm{INT},\ \mathrm{count} : \mathrm{BAG}_s \times s \to \mathrm{INT},\ [\![]\!] : \mathrm{BAG}_s,\ [\![\cdot]\!]^{(\cdot)} : s \times \mathrm{INT} \to \mathrm{BAG}_s,\ \cap, \cup, \uplus : \mathrm{BAG}_s \times \mathrm{BAG}_s \to \mathrm{BAG}_s \bigm| s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\big\}$$
$\mathcal{T}_{\mathrm{BAG}} = \mathcal{T}(\Delta_{\mathrm{BAG}})$ where Δ_{BAG} is the class of standard Σ_{BAG}-algebras.

Σ_{VEC}-theory $\mathcal{T}_{\mathrm{VEC}}$ of vectors

$$\Sigma^{\mathrm{S}}_{\mathrm{VEC}} = \Sigma^{\mathrm{S}}_{\mathrm{INT}} \cup \Sigma^{\mathrm{S}}_{\mathrm{E}} \cup \{\mathrm{VEC}_s \mid s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{VEC}} = \Sigma^{\mathrm{F}}_{\mathrm{INT}} \cup \big\{\mathrm{fst}, \mathrm{end} : \mathrm{VEC}_s \to \mathrm{INT},\ \cdot[\cdot] : \mathrm{VEC}_s \times \mathrm{INT} \to s,\ \mathrm{const} : s \times \mathrm{INT} \times \mathrm{INT} \to \mathrm{VEC}_s,\ \cdot[\cdot{:}\cdot] : \mathrm{VEC}_s \times \mathrm{INT} \times \mathrm{INT} \to \mathrm{VEC}_s,\ \cdot\{\cdot \leftarrow \cdot\} : \mathrm{VEC}_s \times \mathrm{INT} \times s \to \mathrm{VEC}_s \bigm| s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\big\}$$
$$\begin{aligned}\mathcal{T}_{\mathrm{VEC}} = \big\{&\forall u, v : \mathrm{fst}(u) = \mathrm{fst}(v) \wedge \mathrm{end}(u) = \mathrm{end}(v) \wedge \\ &\quad (\forall k : \mathrm{fst}(u) \le k < \mathrm{end}(u) \Rightarrow u[k] = v[k]) \Rightarrow u = v,\\ &\forall x, i, j : \mathrm{fst}(\mathrm{const}(x, i, j)) = i \wedge \mathrm{end}(\mathrm{const}(x, i, j)) = j,\\ &\forall x, i, j, k : i \le k < j \Rightarrow \mathrm{const}(x, i, j)[k] = x,\\ &\forall v, i, j : \mathrm{fst}(v[i{:}j]) = \max(i, \mathrm{fst}(v)) \wedge \mathrm{end}(v[i{:}j]) = \min(j, \mathrm{end}(v)),\\ &\forall v, i, j, k : \mathrm{fst}(v[i{:}j]) \le k < \mathrm{end}(v[i{:}j]) \Rightarrow v[i{:}j][k] = v[k],\\ &\forall v, i, x : \mathrm{fst}(v\{i \leftarrow x\}) = \mathrm{fst}(v) \wedge \mathrm{end}(v\{i \leftarrow x\}) = \mathrm{end}(v),\\ &\forall v, i, x : \mathrm{fst}(v) \le i < \mathrm{end}(v) \Rightarrow v\{i \leftarrow x\}[i] = x,\\ &\forall v, i, x, k : \mathrm{fst}(v) \le k < \mathrm{end}(v) \wedge i \ne k \Rightarrow v\{i \leftarrow x\}[k] = v[k]\big\}\end{aligned}$$

Σ_{BAGOF}-theory $\mathcal{T}_{\mathrm{BAGOF}}$ of bagof function on vectors

$$\Sigma^{\mathrm{S}}_{\mathrm{BAGOF}} = \Sigma^{\mathrm{S}}_{\mathrm{VEC}} \cup \Sigma^{\mathrm{S}}_{\mathrm{BAG}}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{BAGOF}} = \Sigma^{\mathrm{F}}_{\mathrm{VEC}} \cup \Sigma^{\mathrm{F}}_{\mathrm{BAG}} \cup \{\mathrm{bagof} : \mathrm{VEC}_s \to \mathrm{BAG}_s \mid s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\begin{aligned}\mathcal{T}_{\mathrm{BAGOF}} = \big\{&\forall v : |\mathrm{bagof}(v)| = \max(\mathrm{end}(v) - \mathrm{fst}(v), 0),\\ &\forall v : \mathrm{end}(v) - \mathrm{fst}(v) = 1 \Rightarrow \mathrm{bagof}(v) = [\![v[\mathrm{fst}(v)]]\!]^{(1)},\\ &\forall x, i, j : i \le j \Rightarrow \mathrm{bagof}(\mathrm{const}(x, i, j)) = [\![x]\!]^{(j-i)},\\ &\forall v, k : \mathrm{fst}(v) \le k \le \mathrm{end}(v) \Rightarrow\\ &\qquad \mathrm{bagof}(v) = \mathrm{bagof}(v[\mathrm{fst}(v){:}k]) \uplus \mathrm{bagof}(v[k{:}\mathrm{end}(v)])\big\}\end{aligned}$$

Σ_{MAP}-theory $\mathcal{T}_{\mathrm{MAP}}$ of map function on vectors

$$\Sigma^{\mathrm{S}}_{\mathrm{MAP}} = \Sigma^{\mathrm{S}}_{\mathrm{VEC}}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{MAP}} = \Sigma^{\mathrm{F}}_{\mathrm{VEC}} \cup \{f : s \to s' \mid (f{:}s \to s') \in \Sigma^{\mathrm{F}}_{\mathrm{MAP}} \wedge s, s' \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\} \cup \{\mathrm{map}_f : \mathrm{VEC}_s \to \mathrm{VEC}_{s'} \mid (f{:}s \to s') \in \Sigma^{\mathrm{F}}_{\mathrm{MAP}} \wedge s, s' \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\mathcal{T}_{\mathrm{MAP}} = \big\{\forall v : \mathrm{fst}(\mathrm{map}_f(v)) = \mathrm{fst}(v) \wedge \mathrm{end}(\mathrm{map}_f(v)) = \mathrm{end}(v),\ \forall v, k : \mathrm{fst}(v) \le k < \mathrm{end}(v) \Rightarrow \mathrm{map}_f(v)[k] = f(v[k]) \bigm| \mathrm{map}_f \in \Sigma^{\mathrm{F}}_{\mathrm{MAP}}\big\}$$

Fig. 2. Theories of vectors and bags; see Section 3 for details

The theory $\mathcal{T}_{\mathrm{VEC}}$ axiomatises vectors. The first axiom is extensionality, equating all vectors that behave equally under the destructors. The remaining axioms define the constructors (uniquely due to extensionality) in terms of the destructors. Note Σ_{VEC} provides no $\mathrm{append}(\cdot,\cdot)$ because $\mathcal{T}_{\mathrm{VEC}}$ forces vector concatenation to be partial.

Given a signature Σ extending Σ_{VEC}, a Σ-algebra $\mathcal{A}$ is called *vector complete* if for all element sorts $s \in \Sigma_{\mathrm{E}}^{\mathrm{S}}$, all integers i, and all finite sequences $x_0, \ldots, x_{k-1} \in s^{\mathcal{A}}$ there is a vector $v \in \mathrm{VEC}_s^{\mathcal{A}}$ such that $\mathrm{fst}(v)^{\mathcal{A}} = i$ and $\mathrm{end}(v)^{\mathcal{A}} = i + k$ and $v[i + l]^{\mathcal{A}} = x_l$ for all integers l with $0 \leq l < k$. A Σ-theory $\mathcal{T}$ is *vector complete* if every $\mathcal{T}$-satisfiable ground $\hat{\Sigma}$-formula has a vector complete model.

Bagof function. The signature Σ_{BAGOF} extends the union of the signature Σ_{VEC} and Σ_{BAG} with functions $\mathrm{bagof}(\cdot)$ mapping vectors to the multisets of their elements. The theory $\mathcal{T}_{\mathrm{BAGOF}}$ axiomatises these functions. The first axiom equates the length of the argument vector with the cardinality of the resulting multiset. The next two axioms define $\mathrm{bagof}(\cdot)$ for the special cases that the argument vector is of length one or constant. The last axiom admits recursive computation of $\mathrm{bagof}(\cdot)$ by splitting the argument into two subvectors and summing the results.

Map function. The signature Σ_{MAP} extends Σ_{VEC} by adding a set F of unary functions on elements (i. e., $(f{:}s \to s') \in F$ implies $s, s' \in \Sigma_{\mathrm{E}}^{\mathrm{S}}$) and a set F_{map} of map functions on vectors such that $(\mathrm{map}_f : \mathrm{VEC}_s \to \mathrm{VEC}_{s'}) \in F_{\mathrm{map}}$ if and only if $(f{:}s \to s') \in F$. Note that Figure 2 specifies Σ_{MAP} by a fixpoint equation which has infinitely many solutions.[3]

The theory $\mathcal{T}_{\mathrm{MAP}}$ axiomatises the functions map_f, in terms of the vector destructors, thus uniquely defining these functions. Note that $\mathcal{T}_{\mathrm{MAP}}$ does not define the unary functions on elements; these functions are intended to be free.

Base theory. We define the Σ-theory $\mathcal{T}_{\mathrm{BASE}} = \mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}} \cup \mathcal{T}_{\mathrm{VEC}} \cup \mathcal{T}_{\mathrm{MAP}}$ as the union of the above theories excluding $\mathcal{T}_{\mathrm{BAGOF}}$, where $\Sigma = \Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{BAG}} \cup \Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}} \cup \Sigma_{\mathrm{BAGOF}}$ is the union of the above signatures (including Σ_{BAGOF}, i. e., $\mathcal{T}_{\mathrm{BASE}}$ leaves the bagof functions uninterpreted). The following model-theoretic properties will become relevant in Section 5.

Lemma 2. *$\mathcal{T}_{\mathrm{BASE}}$ is vector complete and stably infinite.*

4 Known Decision Procedures Applied to Bags and Vectors

This section employs known results to obtain a decision procedure for ground satisfiability in the combination of the theories of elements, multisets and vectors (including the theory of map functions). We will make repeated use of the following result on the combination of arbitrary theories with free functions.

Proposition 3 (Sofronie-Stokkermans 2005 [9]). *Let $\Sigma' \supseteq \Sigma$ be signatures and let $\mathcal{T}$ be a Σ-theory. If $\mathcal{T}$-satisfiability is decidable for ground $\hat{\Sigma}$-formulae then $\mathcal{T}$-satisfiability is decidable for ground $\hat{\Sigma}'$-formulae.*

[3] Extremal solutions are uninteresting. The least solution would yield $\Sigma_{\mathrm{MAP}} = \Sigma_{\mathrm{VEC}}$, and the greatest solution would likely violate the requirement that Σ_{E} and Σ_{MAP} be disjoint.

The decision procedure behind Proposition 3 reduces a ground $\hat{\Sigma'}$-formula in negation normal form[4] (NNF) to a $\mathcal{T}$-equisatisfiable ground $\hat{\Sigma}$-formula in NNF; the reduction may cause a quadratic blowup.

4.1 Combining the Theories of Elements and Multisets

A decision procedure for the theory $\mathcal{T}_{\mathrm{BAG}}$ of multisets with cardinality is known [7]. We combine this decision procedure with an arbitrary decision procedure for the theory $\mathcal{T}_{\mathrm{E}}$ of elements, using the Nelson-Oppen combination method [6, 11]. This is possible because $\mathcal{T}_{\mathrm{E}}$ and $\mathcal{T}_{\mathrm{BAG}}$ are stably infinite theories (cf. Figure 2 and Lemma 1) over disjoint signatures.

Proposition 4. *Ground $(\mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}})$-satisfiability is decidable.*

4.2 Deciding the Theory of Vectors (Including Map)

We use a decision procedure for the Array Property Fragment [3] to decide ground satisfiability in the union of the theories of vectors and map functions. The procedure reduces the satisfiability problem to ground satisfiability in the combination of the theories of Presburger arithmetic, uninterpreted functions and an unspecified theory of vector elements.

Proposition 5. *Let $\mathcal{T}_0$ be a Σ_0-theory where the signature Σ_0 shares no non-constant function symbols with $\Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}$ except for the function symbols in Σ_{INT}, formally $\Sigma_0 \cap (\Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}) \subseteq \hat{\Sigma_{\mathrm{INT}}}$. Let $\Sigma_1 = \Sigma_0 \cup \Sigma_{\mathrm{INT}} \cup \Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}$ and $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{T}_{\mathrm{INT}} \cup \mathcal{T}_{\mathrm{VEC}} \cup \mathcal{T}_{\mathrm{MAP}}$. If $(\mathcal{T}_0 \cup \mathcal{T}_{\mathrm{INT}})$-satisfiability is decidable for ground $\hat{\Sigma}$-formulae, where Σ extends $\Sigma_0 \cup \Sigma_{\mathrm{INT}}$, then $\mathcal{T}_1$-satisfiability is decidable for ground $\hat{\Sigma_1}$-formulae.*

Proof. Let ϕ be a ground $\hat{\Sigma_1}$-formula (in NNF). Perform the following reductions.

1. Eliminate disequalities and updates: Normalise ϕ w. r. t. the rewrite rules NOTEQ and UPDATE from Figure 3. NOTEQ expresses disequalities $s \neq t$ using extensionality and Skolemisation. UPDATE is based on expressing equations $v = u\{i \leftarrow x\}$ by splitting u and v into three subvectors each (a prefix up to index i, a middle part of length 1 at index i and a suffix from index $i+1$) and equating these accordingly (in particular, equating the middle part of v to a constant vector). The resulting ground $\hat{\Sigma_1}$-formula ϕ' is $\mathcal{T}_1$-equisatisfiable to ϕ but contains no vector disequalities and updates.
2. Purify w. r. t. vector sorts: In a bottom up manner, rewrite $\phi'[t]$ to $\phi'[c] \wedge c = t$, where c is a fresh constant and t a non-constant vector term. The result of normalising ϕ' w. r. t. the above rule is a $\mathcal{T}_1$-equisatisfiable $\hat{\Sigma_1}$-formula ϕ'' such that
   - for all terms of the form $\mathrm{fst}(u)$ or $\mathrm{end}(u)$ or $u[i]$, u is a constant, and
   - all vector atoms are of the form $u = v$ or $v = u[i{:}j]$ or $v = \mathrm{const}(x, i, j)$ or $v = \mathrm{map}_f(u)$, where u and v are constants.

[4] The procedure in [9] expects input in clause form; however, the reduction works just as well for formulae in NNF.

$$[\textsc{NotEq}]\ \frac{\phi\big[u \neq v\big]}{\phi\left[\begin{array}{l}\mathrm{fst}(u) \neq \mathrm{fst}(v) \vee \mathrm{end}(u) \neq \mathrm{end}(v) \vee \\ (\mathrm{fst}(u) \leq k < \mathrm{end}(u) \wedge u[k] \neq v[k])\end{array}\right]}\ \text{if } u, v \text{ vectors} \wedge k \text{ fresh}$$

$$[\textsc{Read}]\ \frac{\phi\big[u[i]\big]}{\phi\big[x\big] \wedge u[i{:}i+1] = \mathrm{const}(x, i, i+1)}\ \text{if } x \text{ fresh}$$

$$[\textsc{Update}]\ \frac{\phi\big[u\{i \leftarrow x\}\big]}{\phi\big[v\big] \wedge \psi(v, u, i, x)}\ \text{if } v \text{ fresh}$$

$$\text{where } \psi(v,u,i,x) \equiv \left\{\begin{array}{l}\big(\mathrm{fst}(u) \leq i < \mathrm{end}(u) \vee u = v\big) \wedge \big(i < \mathrm{fst}(u) \vee \mathrm{end}(u) \leq i \vee \\ \quad \big(\mathrm{fst}(v) = \mathrm{fst}(u) \wedge \mathrm{end}(v) = \mathrm{end}(u) \wedge \\ \quad\ v[\mathrm{fst}(u){:}i] = u[\mathrm{fst}(u){:}i] \wedge v[i{:}i+1] = \mathrm{const}(x, i, i+1) \wedge \\ \quad\ v[i+1{:}\mathrm{end}(u)] = u[i+1{:}\mathrm{end}(u)]\big)\big)\end{array}\right.$$

$$[\textsc{BagOf}]\ \frac{\phi\big[\mathrm{bagof}(u)\big]}{\phi\big[b\big] \wedge b = \mathrm{bagof}(u)}\ \text{if } b \text{ fresh}$$

$$[\textsc{SubConst}]\ \frac{\phi\big[v = u[k{:}l] \wedge u = \mathrm{const}(x, i, j)\big]}{\phi\big[v = \mathrm{const}(x, \max(k, i), \min(l, j)) \wedge u = \mathrm{const}(x, i, j)\big]}$$

$$[\textsc{MapConst}]\ \frac{\phi\big[v = \mathrm{map}_f(u) \wedge u = \mathrm{const}(x, i, j)\big]}{\phi\big[v = \mathrm{const}(f(x), i, j) \wedge u = \mathrm{const}(x, i, j)\big]}$$

Fig. 3. Vector transformations; see sections 4.2 and 5.1 for details

$$[\textsc{Eq}]\ \frac{\phi\big[u = v\big]}{\phi\big[fst_u = fst_v \wedge end_u = end_v \wedge \forall k : fst_u \leq k < end_u \Rightarrow u[k] = v[k]\big]}$$

$$[\textsc{Sub}]\ \frac{\phi\big[v = u[i{:}j]\big]}{\phi\left[\begin{array}{l}fst_v = \max(i, fst_u) \wedge end_v = \min(j, end_u) \wedge \\ \forall k : fst_v \leq k < end_v \Rightarrow v[k] = u[k]\end{array}\right]}$$

$$[\textsc{Const}]\ \frac{\phi\big[v = \mathrm{const}(x, i, j)\big]}{\phi\big[fst_v = i \wedge end_v = j \wedge \forall k : fst_v \leq k < end_v \Rightarrow v[k] = x\big]}$$

$$[\textsc{Map}]\ \frac{\phi\big[v = \mathrm{map}_f(u)\big]}{\phi\big[fst_v = fst_u \wedge end_v = end_u \wedge \forall k : fst_v \leq k < end_v \Rightarrow v[k] = f(u[k])\big]}$$

Fig. 4. Translating to the Array Property Fragment; see Section 4.2 for details

3. Eliminate all subterms of the form $\mathrm{fst}(u)$ and $\mathrm{end}(u)$ in ϕ'' by replacing them with INT-constants fst_u and end_u, respectively, introducing two new INT-constants fst_u, end_u per vector constant u. Then normalise ϕ'' w. r. t. all rewrite rules in Figure 4. This results in a $\mathcal{T}_1$-equisatisfiable $\hat{\Sigma}_1$-formula ϕ''', which falls into the Array Property Fragment [3].

4. Use decision procedure for the Array Property Fragment outlined in [3]:
   - Instantiate universal quantifiers in ϕ'''.
   - Replace all constants u of sort VEC_s by unary functions $f_u : \mathrm{INT} \to s$, and replace all terms of the form $u[i]$ by $f_u(i)$.

   The resulting ground $\hat{\Sigma}$-formula ϕ'''' is $(\mathcal{T}_0 \cup \mathcal{T}_{\mathrm{INT}})$-satisfiable if and only if ϕ''' is $\mathcal{T}_1$-satisfiable, where Σ extends $\Sigma_0 \cup \Sigma_{\mathrm{INT}}$ with the above unary functions f_u and with the unary functions f on element sorts from signature Σ_{MAP}. □

4.3 Deciding the Base Theory

Finally, we pull the results of the previous subsections together to obtain a decision procedure for $\mathcal{T}_{\mathrm{BASE}}$, the union of all theories introduced in Section 3 excluding $\mathcal{T}_{\mathrm{BAGOF}}$. Recall that the signature Σ of $\mathcal{T}_{\mathrm{BASE}}$ includes Σ_{BAGOF}, i. e., $\mathcal{T}_{\mathrm{BASE}}$ treats the bagof functions as free.

Proposition 6. *Ground $\mathcal{T}_{\mathrm{BASE}}$-satisfiability is decidable.*

Proof. Let ϕ be $\hat{\Sigma}$-formula (in NNF).

1. Reduce ϕ to a $\mathcal{T}$-equisatisfiable ground $\hat{\Sigma}'$-formula ϕ' where $\Sigma' = \Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{BAG}} \cup \Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}$, using the decision procedure for free functions (Proposition 3).
2. Reduce ϕ' to a ground $\hat{\Sigma''}$-formula ϕ'' using the decision procedure for vectors (Proposition 5; the Σ_0-theory $\mathcal{T}_0$ there is $\mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}}$ here). The resulting signature Σ'' extends $\Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{BAG}}$ by free unary functions on element sorts (stemming from signature Σ_{MAP}) and free unary functions from INT to element sorts (arising from encoding arrays as unary functions). The formula ϕ'' is $(\mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}})$-satisfiable iff ϕ' is $\mathcal{T}$-satisfiable.
3. Reduce ϕ'' to a $(\mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}})$-equisatisfiable ground $\hat{\Sigma'''}$-formula ϕ''' where $\Sigma''' = \Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{BAG}}$, using the decision procedure for free functions (Proposition 3).
4. Check $(\mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}})$-satisfiability of ϕ''' using the combined decision procedure for elements and multisets (Proposition 4). □

5 A Decision Procedure for Bags, Vectors and Bagof Functions

Recall the Σ-theory $\mathcal{T}_{\mathrm{BASE}}$, defined in Section 3 as the union of all theories excluding $\mathcal{T}_{\mathrm{BAGOF}}$, where Σ is the union of all signatures (including Σ_{BAGOF}). For this section, let $\mathcal{T} = \mathcal{T}_{\mathrm{BASE}} \cup \mathcal{T}_{\mathrm{BAGOF}}$ be the Σ-theory extending $\mathcal{T}_{\mathrm{BASE}}$ with the axioms for the bagof functions.

5.1 Decision Procedure

The decision procedure relies on reducing ground $\mathcal{T}$-satisfiability to ground $\mathcal{T}_{\mathrm{BASE}}$-satisfiability by instantiating axioms of $\mathcal{T}_{\mathrm{BAGOF}}$. The reduction is shown in Figure 5. Termination is obvious. Soundness is established by the lemma below.

Input: Ground $\hat{\Sigma}$-formula ϕ_0 (in NNF).
Output: Ground $\hat{\Sigma}$-formula ϕ_6.
Algorithm:

1. Eliminate definable vector operators, purify and simplify:
   (a) Construct ϕ_1 by normalising ϕ_0 w. r. t. the rule NOTEQ (Figure 3).
   (b) Construct ϕ_2 by normalising ϕ_1 w. r. t. the rules READ, UPDATE and BAGOF (Figure 3).
   (c) Construct ϕ_3 by purifying ϕ_2 w. r. t. vector sorts: In a bottom up manner, rewrite $\phi_2[t]$ to $\phi_2[c] \wedge c = t$, where c is a fresh constant and t a non-constant vector term.
   (d) Construct ϕ_4 by converting ϕ_3 into disjunctive normal form (DNF).
   (e) Construct ϕ_5 by normalising ϕ_4 w. r. t. the rules SUBCONST and MAPCONST (Figure 3).
2. Determine the sets of *vector constants* C, *element terms* E and *index terms* I:

$$\begin{aligned} C &= \{v \mid v \text{ vector constant occurring in } \phi_5\} \\ E &= \{x \mid \exists i, j : \mathrm{const}(x, i, j) \text{ occurs in } \phi_5\} \text{ and} \\ I &= \{\mathrm{fst}(u), \mathrm{end}(u) \mid u \in C\} \cup \\ &\quad \{i, j \mid \exists x : \mathrm{const}(x, i, j) \text{ occurs in } \phi_5 \vee \exists u : u[i{:}j] \text{ occurs in } \phi_5\}. \end{aligned}$$

3. Instantiate (variants of) the $\mathcal{T}_{\mathrm{BAGOF}}$ axioms with terms generated from C, E and I:

$$\phi_6 \equiv \phi_5 \wedge \bigwedge_{u \in C; i,j \in I} \mathrm{Ax}_1^{u,i,j} \wedge \bigwedge_{x \in E; i,j \in I} \mathrm{Ax}_3^{x,i,j} \wedge \bigwedge_{u \in C; i,j,k \in I} \mathrm{Ax}_4^{u,i,j,k}$$

$$\begin{aligned} \text{where} \quad \mathrm{Ax}_1^{u,i,j} &\equiv \mathrm{fst}(u) \le i \le j \le \mathrm{end}(u) \Rightarrow |\mathrm{bagof}(u[i{:}j])| = j - i \\ \mathrm{Ax}_3^{x,i,j} &\equiv i \le j \Rightarrow \mathrm{bagof}(\mathrm{const}(x, i, j)) = [\![x]\!]^{(j-i)} \\ \mathrm{Ax}_4^{u,i,j,k} &\equiv \mathrm{fst}(u) \le i \le k \le j \le \mathrm{end}(u) \Rightarrow \\ &\quad \mathrm{bagof}(u[i{:}j]) = \mathrm{bagof}(u[i{:}k]) \uplus \mathrm{bagof}(u[k{:}j]) \end{aligned}$$

Fig. 5. Reduction to base theory by instantiating $\mathcal{T}_{\mathrm{BAGOF}}$ axioms

Lemma 7 (Soundness). *If ϕ_0 is $\mathcal{T}$-satisfiable then ϕ_6 is $\mathcal{T}_{\mathrm{BASE}}$-satisfiable.*

Proof. As ϕ_0 and ϕ_5 are $\mathcal{T}$-equisatisfiable, it suffices to show that every $\mathcal{T}$-model is a model of the instances $\mathrm{Ax}_1^{u,i,j}$, $\mathrm{Ax}_3^{x,i,j}$ and $\mathrm{Ax}_4^{u,i,j,k}$, for all $u \in C$, $x \in E$ and $i, j, k \in I$.

- $\mathrm{Ax}_1^{u,i,j}$ follows from the first $\mathcal{T}_{\mathrm{BAGOF}}$ axiom (after instantiating v with $u[i{:}j]$) as in $\mathcal{T}_{\mathrm{VEC}}$, $\mathrm{fst}(u) \le i \le j \le \mathrm{end}(u)$ implies $\max(\mathrm{end}(u[i{:}j]) - \mathrm{fst}(u[i{:}j]), 0) = j - i$.
- $\mathrm{Ax}_3^{x,i,j}$ is an instance of the third $\mathcal{T}_{\mathrm{BAGOF}}$ axiom.
- $\mathrm{Ax}_4^{u,i,j,k}$ follows from the fourth $\mathcal{T}_{\mathrm{BAGOF}}$ axiom (after instantiating v with $u[i{:}j]$ and k with k) because in $\mathcal{T}_{\mathrm{VEC}}$, the antecedent $\mathrm{fst}(u) \le i \le k \le j \le \mathrm{end}(u)$ implies $u[i{:}j][\mathrm{fst}(u[i{:}j]){:}k] = u[i{:}k]$ and $u[i{:}j][k{:}\mathrm{end}(u[i{:}j])] = u[k{:}j]$. □

Before we show completeness of the reduction, we point out that step 1 converts the input formula ϕ_0 to a ground DNF formula ϕ_5 such that

- bagof occurs only in atoms of the form $b = \mathrm{bagof}(u)$, where b and u are constants,
- all vector atoms are of the form $u = v$ or $v = u[i{:}j]$ or $v = \mathrm{const}(x, i, j)$ or $v = \mathrm{map}_f(u)$, where u and v are constants,

- all other vector terms are of the form $\mathrm{fst}(u)$ or $\mathrm{end}(u)$, where u is a constant, and
- the arguments of map_f are non-constant, i. e., whenever $\mathrm{map}_f(u)$ occurs in a disjunct ψ then there are no terms x, i and j such that the atom $u = \mathrm{const}(x, i, j)$ would logically follow from ψ in theory $\mathcal{T}_{\mathrm{BASE}}$. Note that this last property is achieved by conversion to DNF and propagation of constant vectors within each disjunct (steps 1d and 1e in Figure 5).

5.2 Completeness in the Absence of Map Functions

We call the signature Σ_{MAP} *trivial* if $\Sigma_{\mathrm{MAP}} = \Sigma_{\mathrm{VEC}}$, i. e., there are no unary functions on elements and no map functions. By model-theoretic arguments, we prove completeness of the reduction shown in Figure 5, given that Σ_{MAP} is trivial.

Lemma 8 (Completeness without map). *Assume Σ_{MAP} trivial. If ϕ_6 is $\mathcal{T}_{\mathrm{BASE}}$-satisfiable then ϕ_0 is $\mathcal{T}$-satisfiable.*

Proof. Assume a $\hat{\Sigma}$-algebra $\mathcal{A}$ which is a $\mathcal{T}_{\mathrm{BASE}}$-model of ϕ_6; w. l. o. g. we assume that $\mathcal{A}$ is vector complete (cf. Lemma 2). It suffices to construct a $\hat{\Sigma}$-algebra $\mathcal{A}'$ which is a $\mathcal{T}$-model of one disjunct ψ of ϕ_5; we assume that $\mathcal{A} \models \psi$.

Recall that C is the set of vector constants occurring in ϕ_5. We choose $\mathcal{A}'$ so that

1. $\mathcal{A}$ and $\mathcal{A}'$ agree on the interpretations of all sorts, all constants except vector constants occurring in ϕ_5, and all function symbols except the bagof functions,
2. $\mathcal{A}'$ interprets $\mathrm{bagof} : \mathrm{VEC}_s \to \mathrm{BAG}_s$ as functions mapping vectors in $\mathrm{VEC}_s^{\mathcal{A}'}$ to the multisets of their elements in $\mathrm{BAG}_s^{\mathcal{A}'}$,
3. $\mathcal{A}'$ interprets vector constants u occurring in ϕ_5 such that $\mathcal{A}$ and $\mathcal{A}'$ agree
   (a) on the interpretations of the ground terms $\mathrm{fst}(u)$ and $\mathrm{end}(u)$, and
   (b) on the interpretations of the ground term $\mathrm{bagof}(u)$.

We have to explain how the interpretations of vector constants can be chosen in such a way that item (3b) holds, i. e., how to keep the interpretations of ground terms $\mathrm{bagof}(u)$ invariant even though the interpretations of the bagof functions change.

Recall the set of index terms I defined in step 2 of the reduction (Figure 5). Let $\langle i_1, \ldots, i_n \rangle$ be an enumeration of I such that $\mathcal{A}$ orders their interpretations in ascending sequence $i_1^{\mathcal{A}} \leq \cdots \leq i_n^{\mathcal{A}}$. Items (1) to (3a) ensure that $\mathcal{A}$ and $\mathcal{A}'$ agree on the interpretations of index terms $i_j \in I$, hence $\mathcal{A}'$ orders their interpretation $i_j^{\mathcal{A}'}$ in the same sequence.

Item (3b) is achieved by an inductive process. Let $j < n$ be minimal such that there is $u \in C$ with $\mathrm{fst}(u)^{\mathcal{A}} \leq i_j^{\mathcal{A}} \leq i_{j+1}^{\mathcal{A}} \leq \mathrm{end}(u)^{\mathcal{A}}$ and $\mathrm{bagof}(u[i_j{:}i_{j+1}])^{\mathcal{A}}$ differing from the multiset of elements in $u[i_j{:}i_{j+1}]^{\mathcal{A}}$. Note that there can be no $x \in E$ — recall the set E of element terms occurring in ϕ_5 — such that $\mathrm{const}(x, i_j, i_{j+1})^{\mathcal{A}} = u[i_j{:}i_{j+1}]^{\mathcal{A}}$. For if there were such $x \in E$ then the $\mathcal{T}_{\mathrm{BAGOF}}$ instance $\mathrm{Ax}_3^{x,i_j,i_{j+1}}$ (appearing as a conjunct in ϕ_6) would ensure that $\mathrm{bagof}(u[i_j{:}i_{j+1}])^{\mathcal{A}}$ equals the multiset of elements in $u[i_j{:}i_{j+1}]^{\mathcal{A}}$. Now let C_u be the set of vector constants whose slice between i_j and i_{j+1} happens to equal $u[i_j{:}i_{j+1}]$ in $\mathcal{A}$, formally

$$C_u = \{v \in C \mid \mathcal{A} \models \mathrm{fst}(v) \leq i_j \leq i_{j+1} \leq \mathrm{end}(v) \wedge u[i_j{:}i_{j+1}] = v[i_j{:}i_{j+1}]\}.$$

Let $\langle x_0, x_1, \ldots, x_{k-1}\rangle$ be an enumeration of the multiset $\mathrm{bagof}(u[i_j{:}i_{j+1}])^{\mathcal{A}}$. Note that the $\mathcal{T}_{\mathrm{BAGOF}}$ instance $\mathrm{Ax}_1^{u,i_j,i_{j+1}}$ constrains the size of the multiset so that $k = i_{j+1}^{\mathcal{A}} - i_j^{\mathcal{A}}$. As $\mathcal{A}$ is vector complete, we can choose the interpretations of all $v \in C_u$ such that for all $l < k$, $v^{\mathcal{A}'}$ stores x_l at index $i_j^{\mathcal{A}'} + l$. This ensures that $\mathcal{A}' \models u[i_j{:}i_{j+1}] = v[i_j{:}i_{j+1}]$. The construction proceeds from there by induction on j.

After the construction is completed, one can show that $\mathcal{A}$ and $\mathcal{A}'$ do in fact agree on the interpretation of $\mathrm{bagof}(u)$, for all $u \in C$. The proof is by induction on the length $\mathrm{end}(u) - \mathrm{fst}(u)$ of u and uses the $\mathcal{T}_{\mathrm{BAGOF}}$ instances $\mathrm{Ax}_4^{u,i,j,k}$, for all $i, j, k \in I$ such that $\mathcal{A} \models \mathrm{fst}(u) \leq i \leq k \leq j \leq \mathrm{end}(u)$.

Obviously, $\mathcal{A}'$ is a model of $\mathcal{T}_{\mathrm{BAGOF}}$ (and thus of $\mathcal{T}$) as that is how the interpretation of the bagof functions was chosen. To show that $\mathcal{A}' \models \psi$, it suffices to show that $\mathcal{A}'$ satisfies every vector atom that $\mathcal{A}$ satisfies (because $\mathcal{A}$ and $\mathcal{A}'$ agree on the interpretation of non-vector literals and all vector literals occurring in ψ are positive). In the case of atoms of the form $v = \mathrm{const}(x, i, j)$ this is so because the construction does not change the interpretation of v. In the case of atoms of the form $u = v$ or $v = u[i{:}j]$, the construction alters the interpretations of corresponding slices of u and v uniformly. □

The decidability of ground satisfiability in the theories of elements, multisets, vectors (excluding map functions) and the bagof function follows from soundness and completeness of the reduction (lemmas 7 and 8) and from decidability of the base theory (Proposition 6).

Theorem 9. *Assume Σ_{MAP} trivial. Then ground $\mathcal{T}$-satisfiability is decidable.*

We remark that the conversion to DNF (step 1d in Figure 5) during the reduction is not necessary if Σ_{MAP} is trivial; NNF is all that's required in that case.

5.3 Completeness in the Presence of Map Functions

To prove completeness of the reduction from Figure 5 when Σ_{MAP} is not trivial, we need syntactic restrictions on the occurrences of map functions in the input formula.

Given a set of element sorts $S \subseteq \Sigma_{\mathrm{E}}^{\mathrm{S}}$, we say a term t is a *S-term* (resp. VEC_S*-term*) if t is a s-term (resp. VEC_s-term) for some $s \in S$. A ground $\hat{\Sigma}$-formula ϕ is *stratified* if there is a partition $\{S_1, \ldots, S_m\}$ of the set of element sorts $\Sigma_{\mathrm{E}}^{\mathrm{S}}$ such that

- for every subterm $\mathrm{map}_f(u)$ of ϕ there are strata S_i and S_{i+1} such that u is a VEC_{S_i}-term and $\mathrm{map}_f(u)$ is a $\mathrm{VEC}_{S_{i+1}}$-term, and
- all arguments of $\mathrm{bagof}(\cdot)$ in ϕ are uniformly VEC_{S_m}-terms.

The verification condition VC from Figure 1 is an example of a stratified formula. Given the strata $S_1 = \{\texttt{String}\}$ and $S_2 = \{\texttt{Resource}\}$, it is easy to check that $\mathrm{map}_{\texttt{MessageResource}}$ maps vectors of strings to vectors of resources, and that all arguments of $\mathrm{bagof}(\cdot)$ are vectors of resources. On the other hand, a formula containing a function symbol $\mathrm{map}_f : \mathrm{VEC}_s \to \mathrm{VEC}_{s'}$ fails to be stratified if $s = s'$, for instance.

Lemma 10 (Completeness for stratified input). *Assume ϕ_0 stratified. If ϕ_6 is $\mathcal{T}_{\mathrm{BASE}}$-satisfiable then ϕ_0 is $\mathcal{T}$-satisfiable.*

Proof (Sketch). Let $S_1, \ldots, S_m$ be the strata for ϕ_0. As stratification is preserved by step 1 of the reduction, ϕ_5 is stratified w. r. t. the same strata. Recall the set C of

vector constants defined in step 2 of the reduction. Stratification induces a partition $\{C_1, \ldots, C_m\}$ of C such that each C_i contains the VEC_{S_i}-constants occurring in ϕ_5. We modify step 3 of the reduction slightly by generating instances of $\mathrm{Ax}_1^{u,i,j}$ and $\mathrm{Ax}_4^{u,i,j,k}$ only for $u \in C_m$.

Now, assume a $\hat{\Sigma}$-algebra $\mathcal{A}$ (which due to Lemma 2 can be assumed vector complete and stably infinite[5]) which is a $\mathcal{T}_{\mathrm{BASE}}$-model of ϕ_6. The construction of a $\mathcal{T}$-model $\mathcal{A}'$ of a disjunct ψ of ϕ_5 is similar to the one in Lemma 8 except for the fact that now $\mathcal{A}'$ may not only change the interpretations of $\mathrm{bagof}(\cdot)$ and of vector constants but also the interpretations of function symbols from signature Σ_{MAP}. The construction proceeds in m phases, yielding a sequence $\langle \mathcal{A}_m, \mathcal{A}_{m-1}, \ldots, \mathcal{A}_1 \rangle$ of $\hat{\Sigma}$-algebras.

The first phase constructs a $\hat{\Sigma}$-algebra $\mathcal{A}_m$ fixing the interpretations of the bagof functions and the vector constants in C_m; this construction is analogous to the proof of Lemma 8. Changing the interpretation some constant $v \in C_m$ may falsify some atom of the form $v = \mathrm{map}_f(u)$. To rectify this, the second phase constructs a $\hat{\Sigma}$-algebra $\mathcal{A}_{m-1}$ fixing the interpretations of vector constants in C_{m-1} (and possibly changing the interpretations of functions in Σ_{MAP}) in order to restore the truth of $v = \mathrm{map}_f(u)$. This in turn may falsify some other map atom, whose truth is restored by constructing $\mathcal{A}_{m-2}$, and so on.

We present the construction of $\mathcal{A}_{m-1}$ in more detail; recall that we assume that $\mathcal{A} \models \psi$, and that ψ is a conjunction of literals. Let $\langle i_1, i_2, \ldots, i_n \rangle$ be the ascending enumeration of index terms as defined in the proof of Lemma 8. Let $j < n$ be minimal such that ψ contains some atom $v = \mathrm{map}_f(u)$ with $\mathcal{A}_m \not\models v[i_j{:}i_{j+1}] = \mathrm{map}_f(u[i_j{:}i_{j+1}])$. Because $\mathcal{A}$ and $\mathcal{A}_m$ essentially differ in the interpretations of vector constants in C_m, we conclude that $v \in C_m$, hence $u \in C_{m-1}$ due to stratification. In $\mathcal{A}_{m-1}$, we change the interpretation of u (and of all u' with $\mathcal{A}_m \models u'[i_j{:}i_{j+1}] = u[i_j{:}i_{j+1}]$) such that the elements of $u[i_j{:}i_{j+1}]^{\mathcal{A}_{m-1}}$ are fresh and pairwise distinct. Freshness means that the elements of $u[i_j{:}i_{j+1}]^{\mathcal{A}_{m-1}}$ occur neither in the $\mathcal{A}$- nor in the $\mathcal{A}_m$-interpretation of any element or vector constant. Because $\mathcal{A}$ and $\mathcal{A}_m$ (which features the same carriers) are stably infinite and vector complete, we can always find enough fresh elements and create arbitrary vectors from them. Next, we change the interpretation of the free function f. Define $f^{\mathcal{A}_{m-1}}$ such that $f^{\mathcal{A}_{m-1}}(u^{\mathcal{A}_{m-1}}[l]) = v^{\mathcal{A}_{m-1}}[l]$, for all integers l with $i_j^{\mathcal{A}_{m-1}} \leq l < i_{j+1}^{\mathcal{A}_{m-1}}$. Due to freshness of the elements in $u[i_j{:}i_{j+1}]^{\mathcal{A}_{m-1}}$, the function $f^{\mathcal{A}_{m-1}}$ is well-defined. The construction proceeds by induction on j.

It is obvious that $\mathcal{A}_{m-1} \models v[i_j{:}i_{j+1}] = \mathrm{map}_f(u[i_j{:}i_{j+1}])$. What remains to be shown is that the construction preserves the truth of other vector atoms occurring in ψ. In the case of atoms of the form $u' = u$ or $u' = u[i{:}j]$, the argument is the same as in the proof of Lemma 8: Both sides are altered uniformly. Finally, the case of atoms of the form $u = \mathrm{const}(x, i, j)$ cannot arise because if it did then step 1e of the reduction would have propagated the constant vector through map_f, replacing the atom $v = \mathrm{map}_f(u)$ with $v = \mathrm{const}(f(x), i, j)$. □

The decidability of satisfiability of stratified ground formulae in the theories of elements, multisets, vectors, map functions and the bagof function follows; the proof is similar to Theorem 9.

[5] By abuse of notation, we call a Σ-algebra $\mathcal{A}$ *stably infinite* if all its carriers are infinite.

Theorem 11. *Ground $\mathcal{T}$-satisfiability is decidable for stratified ground $\hat{\Sigma}$-formulae.*

Relation to Local Theory Extensions. The way the reduction in Figure 5 instantiates universal quantifiers with selected ground terms is reminiscent of local theory extensions [5], and one may wonder whether the theory $\mathcal{T}$ can be viewed as a local extension of the theory $\mathcal{T}_{\mathrm{BASE}}$. However, our model construction does not fit entirely into the framework of local theory extensions because not only does it extend partial extension functions (like the bagof functions) to total ones but also changes the interpretations of base constants and free base functions. It remains to be seen whether the framework of local theory extensions can be suitably generalised to encompass our construction.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein.

References

[1] Aspinall, D., Maier, P., Stark, I.: Monitoring external resources in Java MIDP. Electr. Notes Theor. Comput. Sci. 197(1), 17–30 (2008)
[2] Aspinall, D., Maier, P., Stark, I.: Safety guarantees from explicit resource management. In: Proc. FMCO 2007. LNCS, vol. 5382, pp. 52–71. Springer, Heidelberg (2008)
[3] Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
[4] Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. Ann. Math. Artif. Intell. 50(3-4), 231–254 (2007)
[5] Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
[6] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
[7] Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 218–232. Springer, Heidelberg (2008)
[8] Piskac, R., Kuncak, V.: Linear arithmetic with stars. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 268–280. Springer, Heidelberg (2008)
[9] Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
[10] Suzuki, N., Jefferson, D.: Verification decidability of Presburger array programs. J. ACM 27(1), 191–205 (1980)
[11] Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS, vol. 3229, pp. 641–653. Springer, Heidelberg (2004)
[12] Zarba, C.G.: Combining multisets with integers. In: Voronkov, A. (ed.) CADE 2002. LNCS, vol. 2392, pp. 363–376. Springer, Heidelberg (2002)

A Posteriori Soundness for Non-deterministic Abstract Interpretations*

Matthew Might[1] and Panagiotis Manolios[2]

[1] University of Utah, Salt Lake City, Utah, USA
might@cs.utah.edu
[2] Northeastern University, Boston, Massachusetts, USA
pete@ccs.neu.edu

Abstract. An abstract interpretation's resource-allocation policy (*e.g.*, one heap summary node per allocation site) largely determines both its speed and precision. Historically, context has driven allocation policies, and as a result, these policies are said to determine the "context-sensitivity" of the analysis. This work gives analysis designers newfound freedom to manipulate speed and precision by severing the link between allocation policy and context-sensitivity: abstract allocation policies may be unhinged not only from context, but also from even a predefined correspondence with a concrete allocation policy. We do so by proving that abstract allocation policies can be made non-deterministic without sacrificing correctness; this non-determinism permits precision-guided allocation policies previously assumed to be unsafe. To prove correctness, we introduce the notion of *a posteriori* soundness for an analysis. A proof of *a posteriori* soundness differs from a standard proof of soundness in that the abstraction maps used in an *a posteriori* proof cannot be constructed until *after* an analysis has been run. Delaying construction allows them to be built so as to justify the decisions made by non-determinism. The crux of the *a posteriori* soundness theorem is to demonstrate that a justifying abstraction map can *always* be constructed.

1 Introduction

When engineering a static analysis, better speed and higher precision are principal goals. In abstract interpretation, speed and precision are a function of the abstract allocation policy. By *abstract allocation policy*, we mean the procedure by which an abstract interpretation chooses a resource from a pool of abstract resources during the transition from one abstract state to another. To ground this discussion with some specifics, examples of an abstract resource include abstract environment bindings (for environment analyses [17,18,19,21]), abstract heap addresses (for alias and shape analyses [2,4,22]), abstract contours (for flow analyses [1,15,16,19,23,24,25]), and abstract time-stamps (for frame-string analyses [11,17,18,20]).

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 260–274, 2009.

The abstract allocation policy determines how the abstract state-space partitions the concrete state-space; likewise, a given partitioning uniquely determines an abstract allocation strategy. (More directly, this policy determines the relationship between the concrete and abstract instances of objects like stores, heaps and environments.) A fast, precise analysis needs an allocation policy which summarizes concrete resources that behave alike to the same abstract resource, but which summarizes concrete resources that behave differently to separate abstract resources.

The original motivation for this work came from frustration with context-sensitive policies: for the best result, one must choose the context-sensitive policy whose heuristic is geared toward the behavior of the program under consideration. We wanted to know whether abstract allocation policies could be made sensitive to precision rather than context—whether it is sound to make allocations based purely on precision, or not. Thus, this work begins by asking a general question: what *are* the fundamental, necessary constraints which all abstract allocation policies must obey. Specifically, we want to know whether an abstract allocation policy must directly simulate the concrete allocation policy in order to prove soundness.

Our pursuit of the constraints on abstract allocation policies ends with an unanticipated result: there are no such constraints. To demonstrate completeness, we prove that even the allocation policy which is fully non-deterministic is safe. Central to this result is the criterion of and a proof technique for "*a posteriori* soundness."

1.1 *A Priori* Soundness and Context-Sensitivity

Frequently, abstract resources are selected as a function of the context of the current state, *e.g.*, the current program counter [3], the last k call sites [24], the let-polymorphism of the current function [25], the Cartesian product of argument types [1]. Context has been popular in designing allocation policies because context serves as a reasonably good heuristic for data usage: data structures allocated in the same context (*e.g.*, the same call site, the same stack frame) tend to have similar usage patterns. Context-sensitive policies bleed precision and speed to the extent that they tend to split like resources across several abstract resources, *e.g.*, 1CFA [24], or tend to associate unlike resources as a single abstract resource, *e.g.*, CPA [1].

In an attempt at better performance, one might ask whether allocation policies can be hybridized and proven sound, so that the strengths of the two policies can be combined. For simplistic hybrids, such as the natural "Cartesian product" of two policies, the answer is yes; however, such a hybridization also combines their weaknesses—the splitting tendencies of the two policies multiply: precision goes up, but so does analysis time. If an analysis designer were to compensate for this splitting by making the allocation policy *adaptive*, then proving soundness is suddenly murkier, and the answer *seems* to be no. By *adaptive*, we mean that the allocation policy is allowed to directly consider the ramifications upon the precision or speed of the analysis when selecting an abstract resource to allocate

during a transfer function; adaptive behavior is in contrast to the standard behavior of choosing an abstract resource based on context.

The reason that adaptive behavior seems unsafe is that, under the standard correctness regime, abstract allocation policies have to be a simulation of a concrete allocation policy. Thus, if an abstract allocation policy is informed by the precision of the analysis, that information must be available to the concrete allocation policy, so that the two may remain in sync. In general, of course, this is not possible; the concrete execution has perfect precision, and abstractions of it can have myriad degrees of coarseness.

Example. Consider a concrete store σ and two abstractions thereof, $\hat{\sigma}$ and $\hat{\sigma}'$:

$$\begin{array}{lll} \sigma(1) = 0 & \hat{\sigma}(\hat{\ell}_1) = \{0, 3\} & \hat{\sigma}'(\hat{\ell}_1) = \{0, 3\} \\ \sigma(2) = 3 & & \\ \sigma(3) = 4 & \hat{\sigma}(\hat{\ell}_2) = \{4\} & \hat{\sigma}'(\hat{\ell}_2) = \{4, 5, 7\} \end{array}$$

Suppose that, in an effort to improve precision, an abstract allocation policy always chose the abstract address with the smallest set of abstract values (as opposed to, for instance, picking the label of the current call site). Under this policy, the next abstract address for allocation would be $\hat{\ell}_2$ if the simulation has $\hat{\sigma}$ as its current store, and $\hat{\ell}_1$ if the simulation has $\hat{\sigma}'$ as its current store. It is *impossible* to define a concrete policy that justifies this behavior, because it cannot know whether to pick a concrete address that abstracts to $\hat{\ell}_1$ or to $\hat{\ell}_2$. □

The inability of the abstract allocation policy to deviate from the concrete allocation policy (or, *vice versa*) is embedded in the established process for proving soundness in an abstract interpretation. It is an artifact of the standard process, which is as old as the Cousots' original framework [6,7]. We can abbreviate this soundness process as follows:

1. Define a concrete state-space, L.
2. Define a concrete semantics, $f : L \to L$.
3. Define an abstract state-space, $\hat{L}$.
4. Define an abstraction map, $\alpha : L \to \hat{L}$.
5. Define an abstract semantics, $\hat{f} : \hat{L} \to \hat{L}$.
6. Prove abstract semantics $\hat{f}$ simulates concrete semantics f under map α.

For the duration of this work, we term this process the *a priori* soundness process, because the abstraction map α is constructed before the analysis is run.

1.2 *A Posteriori* Soundness and Non-determinism

This work presents a more flexible soundness process—the *a posteriori* soundness process—in which the abstraction map, and hence the soundness of the analysis, is not constructed until *after* the analysis has been computed. We do so in order to circumvent the excessive strictness that is the byproduct of the *a*

priori soundness process. With *a posteriori* soundness, one can hybridize allocation policies *and* make them adaptive. Most generally, we will be able to make abstract allocation policies non-deterministic; the immediate consequence—that there are no unsound abstract allocation policies—is an unexpected result.

The *a posteriori* soundness process can be summarized as follows:

1. Define a concrete state-space, L.
2. Define a concrete semantics, $f : L \to L$.
3. Define an abstract state-space, $\hat{L}$.
4. Define a non-deterministic abstract semantics, $\hat{f} : \hat{L} \to \mathcal{P}(\hat{L})$.
5. Execute the abstract semantics to produce an abstract transition graph.
6. Construct an abstraction map, $\alpha : L \to \hat{L}$, such that the abstract transition graph simulates the concrete semantics f under the map α.

Proving *a posteriori* soundness then reduces to proving that no matter how the abstract transition graph evolves from the abstract semantics, it is *always* possible to construct a justifying abstraction map.

1.3 Contributions

This work makes the following contributions:

1. The concept of allocation-policy-factored semantics.
2. A framework for non-deterministic abstract interpretation.
3. The correctness proof technique of *a posteriori* soundness.
4. An instance of the framework for higher-order control flow analysis: ∃CFA.
5. A discussion of allocation policies outside the bounds of context-sensitivity.

2 Policy-Factored Concrete Semantics

The goal in this work is to reason about the limits of allocation policies. The first step, then, is to isolate and factor out allocation policies from semantics. Given a concrete state-space Σ, a semantics can be defined by means of a small-step transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$, or congruently, as a transfer function, $g : \Sigma \rightharpoonup \Sigma$. A *policy-factored* transfer function accepts a state and produces a partial function that takes a concrete "locative" from a set L to the next state: $f : \Sigma \to L \rightharpoonup \Sigma$. The set of locatives L denotes a pool of allocatable objects. We use the term *locative* to generalize over entities such as environment bindings, store locations, contours and time-stamps. Given a concrete semantics g and a factored semantics f, the allocation policy is any function $\pi : \Sigma \to L$ constrained so that:

$$g(\varsigma) = f(\varsigma)(\pi(\varsigma)).$$

Example. Consider the one-instruction (Turing-incomplete) language Malloc described by the following grammar:

$$s \in \mathsf{Stmt} ::= lab : var \texttt{ := malloc()}$$

where $lab \in \mathsf{Lab}$ denotes labels on instructions and $var \in \mathsf{Var}$ denotes variables. A state consists of a list of statements, an environment and a store:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Stmt}^* \times \mathit{Env} \times \mathit{Store} \\ \eta \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathbb{N} \\ \sigma \in \mathit{Store} &= \mathbb{N} \rightharpoonup \{0, 1\}. \end{aligned}$$

And the state-to-state transfer function $g : \Sigma \rightharpoonup \Sigma$ just makes allocations:

$$g([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \eta, \sigma) = (\boldsymbol{s}, \eta[var \mapsto n'], \sigma[n' \mapsto 0]),$$

where the address n' is the lowest unused value in the store: $\max(\mathit{dom}(\sigma)) + 1$.

The policy-factored formulation of this semantics is the function f:

$$f([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \eta, \sigma)(\ell) = \begin{cases} (\boldsymbol{s}, \eta[var \mapsto \ell], \sigma[\ell \mapsto 0]) & \ell \notin \mathit{dom}(\sigma) \\ \mathit{undefined} & \ell \in \mathit{dom}(\sigma). \end{cases}$$

Setting $f(\varsigma)(\pi(\varsigma)) = g(\varsigma)$ and solving, we find the set of locatives, $L = \mathbb{N}$, and the allocation policy function $\pi : \Sigma \to L$:

$$\pi(\boldsymbol{s}, \eta, \sigma) = \max(\mathit{dom}(\sigma)) + 1. \qquad \square$$

A concrete semantics must also specify an initial state ς_0. The output of an unfactored semantics is a (possibly infinite) sequence of states $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$, such that $\varsigma_{i+1} = g(\varsigma_i)$. The output of a factored semantics is a (possibly infinite) sequence of states $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$ coupled with a sequence of locatives $\boldsymbol{\ell} = \langle \ell_0, \ell_1, \ldots \rangle$, such that $\varsigma_{i+1} = f(\varsigma_i)(\ell_i)$.

3 Policy-Factored Abstract Semantics

Now that we have created a policy-factored concrete semantics, we can create the machinery central to the subject of this work: the policy-factored abstract semantics. Given an abstract state-space $\hat{\Sigma}$, an abstract semantics can be defined through a transition relation $(\hat{\rightsquigarrow}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, or congruently, an abstract transfer function $\hat{g} : \hat{\Sigma} \to \mathcal{P}\left(\hat{\Sigma}\right)$. Generalizing the abstract transfer function, we can create a policy-factored abstract transfer function, $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$. The factored function takes an abstract state to a *set* of functions; each of these functions takes an abstract locative to a subsequent abstract state.

The factored abstract $\hat{f}$ semantics and the abstract allocation policy function $\hat{\pi} : \hat{\Sigma} \to \hat{L}$ are constrained by the equation:

$$\hat{g}(\hat{\varsigma}) = \bigcup_{\hat{h} \in \hat{f}(\hat{\varsigma})} \hat{h}(\hat{\pi}(\hat{\varsigma})).$$

An abstract semantics, factored or otherwise, must also specify an initial abstract state $\hat{\varsigma}_0 \in \hat{\Sigma}$.

Example. Using the MALLOC language from before, we can create an abstract semantics. An abstract state is a sequence of statements, an abstract environment, and an abstract store:

$$\begin{aligned}\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Stmt}^* \times \widehat{Env} \times \widehat{Store}\\ \hat{\eta} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \mathsf{Lab}\\ \hat{\sigma} \in \widehat{Store} &= \mathsf{Lab} \to \mathcal{P}(\{0,1\}).\end{aligned}$$

A standard abstract-address-per-point, context-sensitive transfer function is the function $\hat{g} : \hat{\Sigma} \to \mathcal{P}\left(\hat{\Sigma}\right)$:

$$\hat{g}([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \hat{\eta}, \hat{\sigma}) = \{(\boldsymbol{s}, \hat{\eta}[var \mapsto lab], \hat{\sigma} \sqcup [lab \mapsto \{0\}])\}.$$

Policy-factoring this function yields $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$:

$$\hat{f}([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \hat{\eta}, \hat{\sigma}) = \{\lambda\hat{\ell}.(\boldsymbol{s}, \hat{\eta}[var \mapsto \hat{\ell}], \hat{\sigma} \sqcup [\hat{\ell} \mapsto \{0\}])\}.$$

(Both $\hat{g}$ and $\hat{f}$ return the empty set for empty statement sequences.) Solving for the abstract allocation policy function $\hat{\pi} : \hat{\Sigma} \to \hat{L}$, we get:

$$\hat{\pi}([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \hat{\eta}, \hat{\sigma}) = lab,$$

and for the set of abstract locatives, we have that $\hat{L} = \mathsf{Lab}$. □

4 Non-deterministic Abstract Interpretation

Factoring out allocation policies makes it possible to describe a framework for abstract interpretation that encompasses all conceivable allocation policies—by making the abstract allocation policy "function" non-deterministic. Of course, the adaptive or precision-sensitive allocation policies we mentioned in the introduction are included under the label *all conceivable*. More specifically, with each application of the transfer function, this framework chooses the abstract locative non-deterministically.

The result of a non-deterministic abstract interpretation is an abstract-locative-labeled transition graph between abstract states:

Definition 1. *An **abstract transition graph** is a labeled graph $(\hat{S}, \rightsquigarrow)$ where:*

- *$\hat{S} \subseteq \hat{\Sigma}$ is a subset of states, and*
- *$(\rightsquigarrow) \subseteq \hat{S} \times \hat{L} \times \hat{S}$ is a set of edges labeled by abstract locatives.*

In lieu of defining an algorithm for computing a non-deterministic abstract interpretation, we define the result of such an abstract interpretation as a *closed* abstract transition graph. A transition graph is closed if it accounts for all abstract transitions from each state:

Definition 2. *An abstract transition graph* $(\hat{S}, \rightsquigarrow)$ *is* ***closed*** *under a policy-factored transfer function* $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$ *iff for each state* $\hat{\varsigma} \in \hat{S}$*, for each state-generator* $\hat{h} \in \hat{f}(\hat{\varsigma})$*, there exists a locative* $\hat{\ell}$ *such that:*

$$\hat{\varsigma} \overset{\hat{\ell}}{\rightsquigarrow} \hat{h}(\hat{\varsigma})(\hat{\ell}).$$

A *least closed graph*, which contains no closed subgraphs, is preferred (but not required) as the result of a static analysis.

With non-deterministic abstract interpretation defined, our new intermediate task is to define simple, liberal criteria that must hold between concrete and abstract policy-factored transfer functions, so that when this condition holds, any closed abstract transition graph represents a simulation of the concrete execution. The next section reviews the standard criteria for the correctness of context-sensitive policies, in preparation for this generalization.

5 The *A Priori* Simulation Criterion

On the road to constructing criteria for policy-factored abstract transfer functions that guarantee simulation, it is illustrative to review the inductive step of an *a priori* proof of soundness. Proving the soundness of an ordinary abstract interpretation reduces to showing that its abstract transfer function simulates the concrete transfer function with respect to some abstraction map $\alpha : \Sigma \to \hat{\Sigma}$. More formally:

Definition 3 (Transfer function simulation). *The abstract transfer function* $\hat{g} : \hat{\Sigma} \to \mathcal{P}\left(\hat{\Sigma}\right)$ ***simulates*** *the concrete transfer function* $g : \Sigma \rightharpoonup \Sigma$ *with respect to the abstraction map* $\alpha : \Sigma \to \hat{\Sigma}$ *iff*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$$

implies

$$\{\alpha(g(\varsigma))\} \sqsubseteq \hat{g}(\hat{\varsigma}).$$

For context-sensitive analyses, it is standard to have a lemma while proving simulation that shows that the abstract allocation policy function is a simulation of the concrete allocation policy function, or more formally:

Definition 4 (Policy simulation). *The abstract policy function* $\hat{\pi} : \hat{\varsigma} \to \hat{L}$ ***simulates*** *the concrete policy function* $\pi : \Sigma \to L$ *with respect to the abstraction maps* $\alpha : \Sigma \to \hat{\Sigma}$ *and* $\alpha_L : L \to \hat{L}$ *iff*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$$

implies

$$\alpha_L(\pi(\varsigma)) \sqsubseteq \hat{\pi}(\hat{\varsigma}).$$

In some frameworks, such as Shivers's formulation of k-CFA [24], this policy simulation lemma is an explicit requirement.

Example. Considering the running MALLOC example again reveals much about how analysis designers tinker with the *concrete* semantics to "engineer" the correctness of their *abstract* allocation policies. With the semantics as formulated, we would have to define a locative abstraction map $\alpha_L : \mathbb{N} \to \mathsf{Lab}$ that can satisfy the policy function simulation requirement. At first glance, this seems awkward—how can we map from a natural number, used as a concrete address, to the label that was used during the abstract interpretation? There doesn't seem to be enough information within the natural number to do so. (Later, we'll show how this can be done with *a posteriori* soundness.) The long-time solution for analysis designers has been to encode the requisite information inside concrete locatives. In this case, the concrete semantics would be modified to use the Cartesian product of labels and naturals for the set of locatives: $L = \mathsf{Lab} \times \mathbb{N}$, yielding:

$$\pi([\![lab : var \texttt{ := malloc()}]\!] : s, \eta, \sigma) = (lab, 1 + \max\{n : (\_, n) \in dom(\sigma)\}).$$

With this reformulation of the concrete semantics, a suitable locative-abstraction map is easily defined:

$$\alpha_L(lab, n) = lab.$$

Now the *a priori* policy-simulation requirement is easy to prove.

A side benefit of proving *a posteriori* soundness is that the concrete semantics do not require reformulation, thereby obviating the need for a proof of equivalence between the original and the re-engineered concrete semantics. □

6 Policy-Factored Abstraction Map

We are closing on our intermediate task of defining a condition on policy-factored semantics that guarantees simulation under non-determinism. Before we can state this condition formally, we need to create a policy-factoring of abstraction maps.

An ordinary proof of soundness for abstract interpretation requires a state-wise abstraction map $\alpha : \Sigma \to \hat{\Sigma}$ to express the relationship between the concrete and abstract domains. In order to allow a non-deterministic abstract interpretation, the proof delays the construction of this map until after the analysis has run. Instead, non-deterministic abstract interpretation employs a policy-factored abstraction map $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$; this function takes an abstraction map over locatives to produce an abstraction map over states.

No further policy-factoring of the semantics is required at this point. Note that the lattice relations and operations on states—$(\sqsubseteq)$, $(\sqcup)$ and $(\sqcap)$—do not require factoring since they operate purely in the abstract state-space.

Example. Returning to the MALLOC example once again, the factored abstraction map on states is $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$:

$$\begin{aligned}\beta(\alpha_L)(\boldsymbol{s},\eta,\sigma) &= (\boldsymbol{s},\beta_{Env}(\alpha_L)(\eta),\beta_{Store}(\alpha_L)(\sigma))\\ \beta_{Env}(\alpha_L)(\eta) &= \lambda var.\alpha_L(\eta(var))\\ \beta_{Store}(\alpha_L)(\sigma) &= \lambda\hat{\ell}.\bigsqcup_{\alpha_L(\ell)\sqsubseteq\hat{\ell}}\{\sigma(\ell)\}.\end{aligned}$$

Dropping in the locative-abstraction map from the previous example yields the expected unfactored abstraction map on states: $\alpha = \beta(\alpha_L)$ □

7 The Dependent Simulation Condition

We finally have the machinery required in order to describe a general, liberal condition under which a non-deterministic abstract interpretation is correct: the dependent simulation condition.

Definition 5. *The policy-factored abstract transfer function* $\hat{f} : \hat{\Sigma} \to \mathcal{P}(\hat{L} \to \hat{\Sigma})$ *is a* ***dependent simulation*** *of the policy-factored concrete transfer function* $f : \Sigma \to L \rightharpoonup \Sigma$ *under the factored abstraction map* $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$ *iff, for all locative abstraction maps* $\alpha_L : L \to \hat{L}$*, if*

$$\beta(\alpha_L)(\varsigma) \sqsubseteq \hat{\varsigma},$$

then for any locative ℓ *and any abstract locative* $\hat{\ell}$*, there exists a state-generator* $\hat{h} \in \hat{f}(\hat{\varsigma})$ *such that:*

$$\beta(\alpha_L[\ell \mapsto \hat{\ell}])(f(\varsigma)(\ell)) \sqsubseteq \hat{h}(\hat{\ell}).$$

8 The *A Posteriori* Soundness Theorem

Having defined the dependent simulation condition, it is now possible to prove that a non-deterministic abstract interpretation satisfying this condition is correct, thereby demonstrating that there is no such thing as an illegal abstract allocation policy. A standard proof of soundness is not possible in this case: the abstract allocation policy must simulate the concrete allocation policy, but we cannot describe the abstraction map in advance when the abstract policy is non-deterministic.

First, we must define the concept of a sound simulation for abstract transition graphs over concrete executions.

Definition 6. *An abstract transition graph* $(\hat{S}, \rightsquigarrow)$ *is a* ***sound simulation*** *of a sequence of states* $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$ *under the abstraction map* $\alpha : \Sigma \to \hat{\Sigma}$ *iff*

- *for each* $i \leq length(\boldsymbol{\varsigma})$*:*
$$\{\alpha(\varsigma_i)\} \sqsubseteq \hat{S}, \text{ and}$$
- *for each* $i < length(\boldsymbol{\varsigma})$*:*
$$\{(\alpha(\varsigma_i), \alpha(\varsigma_{i+1}))\} \sqsubseteq (\rightsquigarrow).$$

In other words, each concrete state and each concrete transition is represented in the abstract graph by an abstract state and an abstract edge.

Next, we prove the *a posteriori* soundness theorem. It states that, when the dependent simulation condition is met, a locative-abstraction map that makes a closed abstract transition graph a simulation of a concrete execution must exist.

Theorem 1 (*A posteriori* soundness). *If:*

- $(\boldsymbol{\varsigma}, \boldsymbol{\ell})$ *is a concrete execution for factored transfer function* f*, and*
- $\hat{f}$ *is a dependent simulation of* f *under factored map* β*, and*
- $(\hat{S}, \rightsquigarrow)$ *is a closed abstract transition graph for* $\hat{f}$ *where* $\hat{\varsigma}_0 \in \hat{S}$*, and*
- *for all maps* $\alpha_L : L \to \hat{L}$*,* $\beta(\alpha_L)(\varsigma_0) \sqsubseteq \hat{\varsigma}_0$*,*

then there exists a map $\alpha_L : L \to \hat{L}$ *such that the graph* $(S, \rightsquigarrow)$ *is a sound simulation of the sequence* $\boldsymbol{\varsigma}$ *under the abstraction map* $\beta(\alpha_L)$*.*

Proof. The proof proceeds by construction of the locative abstraction map. We do so by defining a sequence of abstract states $\hat{\boldsymbol{\varsigma}} = \langle \hat{\varsigma}_0, \hat{\varsigma}_1, \ldots \rangle$, a sequence of abstract locatives $\hat{\boldsymbol{\ell}} = \langle \hat{\ell}_0, \hat{\ell}_1, \ldots \rangle$ and a sequence of partial locative abstraction maps $\boldsymbol{\alpha} = \langle \alpha_0, \alpha_1, \ldots \rangle$ through recurrence equations. We show by induction that $\beta(\alpha_i)(\varsigma_i) \sqsubseteq \hat{\varsigma}_i$.

Let N be the length of the concrete execution sequence $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$. For later use, fix a choice function $\mathit{choose} : \mathcal{P}\left(\hat{L} \times \hat{\Sigma}\right) \to (\hat{L} \times \hat{\Sigma})$. The initial abstraction map α_0 is defined to be $\bot_{\hat{L}}$ at every point: $\alpha_0 = \lambda \ell . \bot_{\hat{L}}$.

We construct the abstract locative $\hat{\ell}_i$ and the abstract state $\hat{\varsigma}_{i+1}$ simultaneously. Let the set of candidate transitions $C_i \subseteq \hat{L} \times \hat{\Sigma}$ be:

$$C_i = \{(\hat{\ell}, \hat{\varsigma}) : \hat{\varsigma}_i \overset{\hat{\ell}}{\rightsquigarrow} \hat{\varsigma} \text{ and } \beta(\alpha_i[\ell_i \mapsto \hat{\ell}])(\varsigma_i) \sqsubseteq \hat{\varsigma}\}.$$

The set C_i must be non-empty because the graph is closed and the dependent simulation criterion is satisfied. So, we set $(\hat{\ell}_i, \hat{\varsigma}_{i+1}) = \mathit{choose}(C_i)$ and $\alpha_{i+1} = \alpha_i[\ell_i \mapsto \hat{\ell}_i]$. The satisfying locative abstraction map is then:

$$\alpha_L = \lim_{i \to N} \alpha_i.$$

Example. We will now construct *a posteriori* locative-abstraction maps for the following Malloc program:

```
L1: x := malloc()
L2: y := malloc()
L3: z := malloc()
```

Using natural numbers for concrete addresses yields the following final state:

$$\varsigma_f = (\langle\rangle, [[\![\mathtt{x}]\!] \mapsto 1, [\![\mathtt{y}]\!] \mapsto 2, [\![\mathtt{z}]\!] \mapsto 3], [1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0]).$$

If a non-deterministic abstract interpretation allocated the abstract locative $\hat{\ell}_1$, then $\hat{\ell}_2$, then $\hat{\ell}_2$, then the locative-abstraction map $\alpha_L : L \to \hat{L}$ would be:

$$\alpha_L(1) = \hat{\ell}_1 \qquad \alpha_L(2) = \hat{\ell}_2 \qquad \alpha_L(3) = \hat{\ell}_2.$$

If, instead, it had allocated $\hat{\ell}_1$, then $\hat{\ell}_2$, then $\hat{\ell}_1$, then the locative-abstraction map would be:

$$\alpha_L(1) = \hat{\ell}_1 \qquad \alpha_L(2) = \hat{\ell}_2 \qquad \alpha_L(3) = \hat{\ell}_1.$$

In each case, the locative-abstraction map leads to simulation. □

9 Example: ∃CFA

To demonstrate the applicability of the *a posteriori* soundness theorem, we construct the generalized, non-deterministic higher-order control-flow analysis (∃CFA) by policy-factoring k-CFA [23,24]. This leads to a factored concrete transfer function, $f : \Sigma \to L \rightharpoonup \Sigma$, a factored abstract transfer function, $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$ and a factored abstraction map, $\beta : (L \to \hat{L}) \to (\Sigma \to \hat{\Sigma})$. An interesting side effect of constructing ∃CFA is that it doubles as a proof of correctness for all existing CFAs (*e.g.*, 0CFA, k-CFA, poly/CFA, CPA) and all future CFAs.

For simplicity, we operate over continuation-passing style (CPS), as described in the following grammar:

$$\begin{aligned} v &\in \mathsf{Var} \quad \text{is a set of identifiers} \\ \lambda &\in \mathsf{Lam} ::= (\lambda\ (v_1 \cdots v_n)\ call) \\ f, e &\in \mathsf{Exp} \ = \ \mathsf{Var} + \mathsf{Lam} \\ call &\in \mathsf{Call} ::= (f\ e_1 \cdots e_n). \end{aligned}$$

A concrete state consists of a call site, a binding environment over variables and a value environment over bindings:

$$\begin{aligned} \varsigma \in \Sigma_{\text{CPS}} &= \mathsf{Call} \times \mathit{BEnv} \times \mathit{VEnv} \\ \\ \rho \in \mathit{BEnv} &= \mathsf{Var} \rightharpoonup L \\ b \in \mathit{Bind} &= \mathsf{Var} \times L \\ \\ ve \in \mathit{VEnv} &= \mathit{Bind} \rightharpoonup D \\ d \in D &= \mathit{Clo} \\ clo \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{BEnv}. \end{aligned}$$

The policy-factored concrete transfer function f is:

$$f([\![(f\ e_1 \cdots e_n)]\!], \rho, ve)(\ell) = (call, \rho'', ve'),$$

defined only if no variable v exists so that $(v, \ell) \in \mathit{dom}(ve)$ and where:

$$\begin{aligned} ([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \rho') &= \mathcal{A}(f, \rho, ve) \\ d_i &= \mathcal{A}(e_i, \rho, ve) \\ \rho'' &= \rho'[v_i \mapsto \ell] \\ ve' &= ve[(v_i, \ell) \mapsto d_i], \end{aligned}$$

where the argument evaluator is $\mathcal{A} : \mathsf{Exp} \to \mathit{BEnv} \rightharpoonup D$:

$$\begin{aligned}\mathcal{A}(\lambda, \rho, ve) &= (\lambda, \rho)\\ \mathcal{A}(v, \rho, ve) &= ve(v, \rho(v)).\end{aligned}$$

The abstract state-space is:

$$\begin{aligned}\hat{\varsigma} \in \hat{\Sigma}_{\mathrm{CPS}} &= \mathsf{Call} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{VEnv}}\\ \\ \hat{\rho} \in \widehat{\mathit{BEnv}} &= \mathsf{Var} \rightharpoonup \hat{L}\\ \hat{b} \in \widehat{\mathit{Bind}} &= \mathsf{Var} \times \hat{L}\\ \\ \widehat{ve} \in \widehat{\mathit{VEnv}} &= \widehat{\mathit{Bind}} \to \hat{D}\\ \hat{d} \in \hat{D} &= \mathcal{P}\left(\widehat{\mathit{Clo}}\right)\\ \widehat{clo} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{BEnv}}.\end{aligned}$$

According to this, the abstract locatives correspond to the abstract contours of higher-order control-flow analysis.

The policy-factored abstract transfer function $\hat{f}$ is:

$$\hat{f}([\![(f\ e_1 \cdots e_n)]\!], \hat{\rho}, \widehat{ve}) = \{\lambda\hat{\ell}.\mathcal{F}(\widehat{clo})(\hat{\ell}) : \widehat{clo} \in \hat{\mathcal{A}}(f, \hat{\rho}, \widehat{ve})\},$$

where the state finalizer $\mathcal{F} : \widehat{\mathit{Clo}} \to \hat{L} \to \hat{\Sigma}$ is:

$$\begin{aligned}\mathcal{F}([\![(\lambda\ (v_1 \cdots v_n)\ \mathit{call})]\!], \hat{\rho}')(\hat{\ell}) = (\mathit{call}, \hat{\rho}'', \widehat{ve}'), \text{ where:}\\ \hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\rho}, \widehat{ve})\\ \hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{\ell}]\\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \hat{\ell}) \mapsto \hat{d}_i],\end{aligned}$$

where the abstract argument evaluator is $\hat{\mathcal{A}} : \mathsf{Exp} \to \widehat{\mathit{BEnv}} \to \hat{D}$:

$$\begin{aligned}\hat{\mathcal{A}}(\lambda, \hat{\rho}, \widehat{ve}) &= \{(\lambda, \hat{\rho})\}\\ \hat{\mathcal{A}}(v, \hat{\rho}, \widehat{ve}) &= \widehat{ve}(v, \hat{\rho}(v)).\end{aligned}$$

The appropriate policy-factored abstraction map $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$ walks component-wise over the state-space:

$$\begin{aligned}\beta(\alpha_L)(\mathit{call}, \rho, ve) &= (\mathit{call}, \beta_{\mathit{BEnv}}(\alpha_L)(\rho), \beta_{\mathit{VEnv}}(\alpha_L)(ve))\\ \beta_{\mathit{BEnv}}(\alpha_L)(\rho)(v) &= \alpha_L(\rho(v))\\ \beta_{\mathit{VEnv}}(\alpha_L)(ve))(v, \hat{\ell}) &= \bigsqcup_{\alpha_L(\ell)=\hat{\ell}} \{\beta_{\mathit{Clo}}(\alpha_L)(ve(v, \ell))\}\\ \beta_{\mathit{Clo}}(\alpha_L)(\lambda, \rho) &= (\lambda, \beta_{\mathit{BEnv}}(\alpha_L)(\rho)).\end{aligned}$$

10 Adaptive Allocation Policies

One of the payoffs for proving all conceivable allocation policies correct is in the ability to make allocation policies *adaptive*, or more precisely, precision-sensitive. A precision-sensitive abstract allocation policy makes allocation decisions based on the perceived effect upon the precision of the analysis.

One way to make a context-sensitive allocation policy into a greedy precision-sensitive policy is to augment it with a reserve pool of m abstract locatives, where m is a fixed cap. Abstract locatives in the reserve pool are not associated with a particular context; they are allocated as necessary to prevent excessive merging. For example, if the default abstract locative to be allocated would cause a closure over λ_{42} to coexist in the same abstract store slot as a closure over λ_{314}, then an adaptive analysis can allocate from the reserve pool of abstract locatives to prevent the merge, so long as the reserve pool is not yet exhausted. Adding a reserve pool of abstract locatives to an existing context-sensitive analysis is a simple way to alleviate the damage to precision from places where the context-sensitive heuristic causes excess merging. Enlarging the reserve pool is an effective way of gradually improving the precision of the analysis.

Adaptive analysis alleviates excess splitting by looking for abstract locatives which, upon allocation, cause the least change to the abstract store. For example, if the default abstract locative would give a fresh slot to a closure over λ_{42}, when another slot already contains a closure over λ_{42}, the adaptive analysis may opt to allocate the locative already in use.

Adaptive allocation policies, which are not provably correct under *a priori* soundness, highlight the practical advantages of non-determinism in abstract interpretation.

11 Related Work

This work taps into the foundations of the Cousots' work on abstract interpretation [6,7]. The standard soundness recipe we presented is a simplification of the soundness regime presented throughout their work [5,8]. The use of *a posteriori* abstraction maps is a simple way of extending their framework to allow a practical degree of non-determinism in abstract interpretation.

This work should not be confused with the body of work on the (deterministic) abstract interpretation of non-deterministic systems [9,13]. However, it is likely that non-deterministic abstract interpretation of non-deterministic systems will lead to considerable gains in precision. This work should also not be confused with random interpretation [10], which is unsound. Our work is related in that we enable probabilistic abstract interpretation, but our work retains soundness.

This work impacts the large body of work on alias and shape analysis [2,3,4,12,22] by liberating these analyses from the needless rigidity imposed by *a priori* abstraction maps.

This work also directly impacts higher-order relatives of alias and shape analysis, environment analysis [14,17,18,19,21] and control-flow analysis [15,16,23,24], by expanding the set of contour-allocation schemes.

12 Conclusions and Future Work

We have presented a framework for enabling sound non-deterministic abstract interpretations. We introduced non-determinism into allocation policies in order to free analyses from the rigidity of *a priori* abstraction maps. By proving the correctness of the non-deterministic framework using the novel proof technique of *a posteriori* abstraction maps, we have proven that all conceivable abstract allocation policies are correct. We discussed a practical benefit: that allocation policies may be made adaptive with respect to analytic precision, a behavior which cannot be proven sound under the Cousots' standard correctness framework. And, we instantiated this framework to create a non-deterministic flow analysis: ∃CFA.

For future work, we plan to explore precision-sensitive allocation where abstract locatives are allocated probabilistically, according to evolving distributions that tend toward "do not allocate" in the limit. We also plan to investigate the issue of optimality. For example, for an alias analysis, a good metric would be the average size of an abstract value set in the abstract store; the equivalent metric for a CFA would be the average size of a flow set. For a fixed set of n abstract locatives and a given program, there must exist optimal allocation policies which minimize this metric. With a notion of optimality, we can begin to ask whether there are fundamental bounds on precision, and whether an optimal allocation policy can be computed without resorting to exhaustive search.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (May 1994)
3. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
4. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of Pointers and Structures. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, White Plains, New York, June 1990, pp. 296–310 (1990)
5. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Electronic Notes in Theoretical Computer Science 6, 25 pages (1997), `http://www.elsevier.nl/locate/entcs/volume6.html`
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282. ACM Press, New York (1979)
8. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)

9. Gallagher, J., Gallagher, J.P., Puebla, G., Puebla, G.: Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 243–261. Springer, Heidelberg (2002)
10. Gulwani, S.: Program analysis using random interpretation. Ph.D. Dissertation, UC-Berkeley (2005)
11. Harrison, W.L.: The interprocedural analysis and automatic parallelization of Scheme programs. Lisp and Symbolic Computation 2(3/4), 179–396 (1989)
12. Hudak, P.: A semantic model of reference counting and its abstraction (detailed summary). In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, Cambridge, Massachusetts, August 1986, pp. 351–363 (1986)
13. Huth, M.: An abstraction framework for mixed non-deterministic and probabilistic systems. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 419–444. Springer, Heidelberg (2004)
14. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.K.: Single and loving it: Must-alias analysis for higher-order languages. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, January 1998, pp. 329–341 (1998)
15. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 114–128. Springer, Heidelberg (1981)
16. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 66–74. ACM, New York (1982)
17. Might, M.: Environment Analysis of Higher-Order Languages. PhD thesis, Georgia Institute of Technology (2007)
18. Might, M., Shivers, O.: Environment analysis via ΔCFA. In: Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages (POPL 2006), Charleston, South Carolina, January 2006, pp. 127–140 (2006)
19. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006), Portland, Oregon, September 2006, pp. 13–25 (2006)
20. Might, M., Shivers, O.: Analyzing the environment structure of higher-order languages using frame strings. Theoretical Computer Science 375(1–3), 137–168 (2007)
21. Might, M., Shivers, O.: Exploiting reachability and cardinality in abstract interpretation. Journal of Functional Programming (2008)
22. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Symposium on Principles of Programming Languages, pp. 105–118 (1999)
23. Shivers, O.: Control-flow analysis in Scheme. In: Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, June 1988, pp. 164–174 (1988)
24. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU-CS-91-145 (May 1991)
25. Wright, A.K., Jagannathan, S.: Polymorphic splitting: An effective polyvariant flow analysis. ACM Transactions on Programming Languages and Systems 20(1), 166–207 (1998)

An Automata-Theoretic Dynamic Completeness Criterion for Bounded Model-Checking

Rotem Oshman

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract. Bounded model-checking is a technique for finding bugs in very large designs. Bounded model-checking by itself is incomplete: it can find bugs, but it cannot prove that a system satisfies a specification. A *dynamic completeness criterion* can allow bounded model-checking to prove properties. A dynamic completeness criterion typically searches for a "beginning" of a bug or bad behavior; if no such "beginning" can be found, we can conclude that no bug exists, and bounded model-checking can terminate. Dynamic completeness criteria have been suggested for several temporal logics, but most are tied to a specific bounded model-checking encoding, and the ones that are not are based on nondeterministic Büchi automata. In this paper we develop a theoretic framework for dynamic completeness criteria based on alternating Büchi automata. Our criterion generalizes and explains several existing dynamic completeness criteria, and is suitable for both linear-time and universal branching-time logic. We show that using alternating automata rather than nondeterministic automata can lead to much smaller completeness thresholds.

1 Introduction

Bounded model-checking (BMC) is a model-checking method that has gained popularity due to its ability to handle large industrial designs [4],[5]. Bounded model-checking is an iterative process in which one searches for a bug of increasing bounded length. In each iteration, one searches for a bug of size k, by constructing a Boolean formula which is satisfiable iff such a bug exists. A SAT solver is then used to determine whether or not the formula is satisfiable. If it is, then a bug has been found; otherwise, one increases the bound k and searches for a bug of greater size.

There are many BMC encodings for various fragments of linear-time logic and automata on words; e.g., [4] for LTL, [10] for PLTL, [11] for weak alternating Büchi automata. Many are based, directly or indirectly, on the idea of constructing a product automaton $M \times A$ for the model M in question and an automaton A which describes all the undesirable behaviors. Any accepting run of the product automaton $M \times A$ corresponds to a bad behavior of the model; thus, to check if the model contains a bug, we can search for an accepting run of the product automaton. Using automata as specification mechanisms can lead to simple and generic encodings. Even encodings based on temporal logic (e.g., [10] and [12]) can often be viewed as simulating the run of the product automaton, although they do not construct it directly.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 275–289, 2009.

Bounded model-checking is typically a semi-decision procedure: it is able to find bugs, but not to prove the correctness of properties. A *completeness threshold* is an upper bound on the size of k, such that if no bug has been found when the bound reaches k, then no bug exists. Thus, if a completeness threshold for the property and model in question is known, bounded model-checking can halt if the completeness threshold is reached and no bug was found, and conclude that the model satisfies the property.

Completeness thresholds can be broadly divided into two classes, although the division is not clear-cut. *Static* completeness thresholds ([5], [6]) attempt to over-approximate the size of the "longest shortest bug" the system can contain. For example, if a model does not satisfy an invariant p, then there exists a shortest path from an initial state of the model to a state that does not satisfy p. A static completeness threshold for invariant properties is therefore given by the length of the longest shortest path in the model (the *diameter*).

In contrast, *dynamic* completeness thresholds are based on a *dynamic completeness criterion*, which attempts to determine whether the current bound is already large enough to allow full exploration of the relevant part of the model. Dynamic completeness criteria typically check for the existence of a "beginning" of a counter-example (or bug). If such a beginning of size k cannot be found, then there cannot exist a counter-example of size greater than k, and there is no need to increase the bound. For example, the LTL property $\varphi = pUq$ describes a path in which q holds at some point, and until that point, p holds. Suppose φ describes the bad behaviors of a system. A dynamic completeness criterion for φ might check if there exists a simple (loop-free) path of length k, such that all states along the path are labeled with p. Such a path represents a "beginning" of a witness for φ. If we cannot find a witness of length k for φ, and we cannot find a simple path of length k as described above, then there cannot exist a witness of length greater than k for φ. Therefore, in this case bounded model-checking can terminate and conclude that the system contains no path that satisfies φ.

The effectiveness of dynamic completeness criteria has been shown in experimental results ([10], [22], [24]). However, designing completeness criteria that are both sound and effective can be challenging. For instance, the completeness criterion in [22] contains a subtle flaw: a constraint introduced to cause earlier termination and increase the effectiveness of the criterion causes the criterion to be unsound. For details, see [17]. In addition, existing completeness criteria are often custom-designed to fit one particular encoding. For example, the dynamic completeness criteria of [22], [18] and [24] are all based on ideas similar to the ones on which the current paper is based, but they each develop the completeness criterion anew to fit the particular encoding.

In this work we present an automata-theoretic dynamic completeness criterion for alternating Büchi automata. Our criterion generalizes several existing completeness criteria by formalizing the notion of a "beginning" of an accepting computation. The criterion we suggest is independent of a particular encoding; in addition to serving as a theoretical framework for existing completeness criteria, it can be instantiated to fit automata-based BMC encodings for which there is currently no dynamic completeness criterion, such as [11].

To our knowledge, the criterion we suggest is the first completeness criterion that can handle alternating automata. The choice of alternating Büchi automata as a specification mechanism is motivated by two factors. First, alternating Büchi automata are powerful enough to express all ω-regular properties. [11] developed an encoding for weak alternating Büchi automata, and showed that the increased expression power did not carry significant performance penalties.

The second factor is the succintness of alternating automata. It is well-known that an alternating automaton can be exponentially smaller than any equivalent nondeterministic automaton [20]. In this paper we show another compelling reason to use alternating automata as a specification mechanism: they can have much smaller completeness thresholds than the corresponding nondeterministic automata. This result is related to, but does not follow directly from, the exponential gap in the number of states.

In addition to linear-time logic, several BMC encodings and accompanying completeness criteria have been suggested for universal branching-time logic ([19], [23], [22], [18]). Our completeness criterion is based on automata on infinite words, which express linear-time properties. However, the criterion is also applicable to universal automata on infinite trees, which express universal branching-time properties. This is because our criterion is based on the product of the model and the automaton. The product of a model and a alternating automaton on infinite trees is an alternating automaton on infinite words [14]; thus, our dynamic completeness criterion, which is based on alternating automata on infinite words, is applicable to branching-time logic as well. Note, however, that in a branching-time setting, a Büchi acceptance condition is not expressive enough to express all ω-regular tree properties. Our criterion can therefore only handle the alternation-free fragment of universal μ-calculus.

The rest of the paper is organized as follows. In Section 3 we review the automata-theoretic approach to linear-time logic and define notation and terminology. In Section 4 we present the dynamic completeness criterion and the resulting completeness threshold. We show that the criterion is sound, and characterize its completeness. In Section 5 we show that there is an exponential ratio between the completeness thresholds of alternating and nondeterministic automata. We conclude in Section 6.

2 Related Work

In the original work on BMC ([4], [5]), the diameter of the model is suggested as a completeness threshold for formulas of the form EFp ("p is reachable"). [5] also shows a pessimistic completeness threshold of $|M| \times 2^{|\varphi|}$ for general LTL formulas φ. In [6], tighter completeness thresholds are shown for various classes of temporal properties, among them the class of all ω-regular properties, based on automata-theoretic methods. The completeness threshold suggested in [6] for general ω-regular properties is an over-approximation of the length of the shortest lasso-shaped accepting run of the product automaton. Our own work is based on similar ideas; however, the automata we consider are alternating, while [6] bases its threshold on nondeterministic automata. As we show in Section 5, using nondeterministic automata as a specification mechanism can increase the

completeness threshold significantly. In addition, the completeness threshold of [6] does not take the form of a *dynamic completeness criterion* which is evaluated at different bounds to determine whether or not the completeness threshold has been reached. In [2], the authors apply similar ideas to [6], this time in a form closer to a dynamic completeness criterion: to check whether the completeness threshold has been reached, one can check the satisfiability of several Boolean formulas, which roughly speaking describe the existence of loop-free fragments of an accepting run in the product automaton. Unlike our own work, the completeness criteria of [2] check for the existence of both a "beginning" and an "ending" of an accepting run (forward and backward traversal). However, [2] is still restricted to nondeterministic automata. In [3], the authors of [2] extend their termination criterion to *generalized* nondeterministic Büchi automata, in which the acceptance criterion can consist of several accepting sets, and show that using generalized automata can lead to smaller completeness thresholds. The completeness threshold we suggest in this paper is easily extended to generalized Büchi automata.

In [10], an incremental encoding is presented for PLTL, along with a dynamic completeness criterion based on the idea of searching for a "beginning" of a witness. In an incremental scheme, the encoding is composed of two parts – a k-invariant part, containing constraints that are retained when the bound is increased, and a k-dependent part containing constraints that are discarded when the bound is increased. The formula used in [10] to determine whether the completeness threshold has been reached is obtained from the formula used to search for a witness by removing the k-dependent constraints and adding a simple-path constraint. Removing the k-dependent constraints has the effect of dropping eventuality requirements (e.g., when searching for a witness for Fp, the requirement that p be satisfied at some point along the path is a k-dependent constraint). The completeness formula of [10] is highly specific to the incremental scheme and the particular encoding used in [10]. Our completeness criterion can be extended to handle temporal logic with past operators by extending it to two-way automata on words [21].

Several bounded model-checking encodings have been suggested for universal branching-time temporal logic [18], [19], [22], [23]. [22] and [18] show accompanying dynamic completeness criteria for their respective encodings, and a dynamic completeness criterion for the encoding of [19] is presented in [24]. The criteria of [22], [18] and [24] are again highly encoding-specific, and all use a similar idea of searching for a "beginning" of a witness.

A related SAT-based technique which can prove properties is temporal induction [7], which can prove invariants. General safety and liveness properties can be transformed into invariants, but such translations increase the size of the model and may increase the depth necessary for bounded model-checking.

Our work is also closely related to [8], which discusses extensions of LTL that can be used to reason about *truncated* paths. Our notion of a partial run corresponds to the weak semantics of LTL for truncated paths described in [8], and can be taken as an automata-theoretic formulation of the weak semantics. We are interested in investigating this connection.

3 Preliminaries

Given a set X, we denote by $\mathcal{B}^+(X)$ the set of positive Boolean formulas obtained by applying the connectives $\wedge$ (conjunction) and $\vee$ (disjunction) to elements of X, as well as the formulas **true** and **false**. We say that a subset $Y \subseteq X$ *satisfies* a formula $\alpha \in \mathcal{B}^+(X)$, and denote $Y \models \alpha$, if the assignment v_Y, defined by $v_Y(x) = 1$ if $x \in Y$ and $v_Y(x) = 0$ if $x \notin Y$, satisfies the formula α.

An *alternating Büchi automaton on infinite words* is a tuple $A = (\Sigma, Q, q_0, \delta, F)$, where Σ is the automaton's alphabet, Q is the set of automaton states, $q_0 \in Q$ is the initial state of the automaton, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is a transition relation, and $F \subseteq Q$ is the set of *accepting* (or *fair*) states. A *nondeterministic* automaton is an alternating automaton which has only disjunctions in all its transitions. We use Σ^ω to denote the set of infinite words over the alphabet Σ, and we use x^ω to denote the infinite word obtained by iterating the finite word x infinitely often.

To model the runs of A we use Q-trees. A *Q-tree* is a pair $t = (N, \ell)$, where $N \subseteq \mathbb{N}^*$ is a prefix-closed set of tree nodes, and $\ell : N \to Q$ labels each node of the tree with an automaton state. The root of the tree is the empty word ε, and given a node $n \in N$, the set of children of n in the tree is given by $children(n) = \{n' \mid n' = n \cdot i \text{ for some } i \in \mathbb{N}\}$. We denote by $|n|$ the length of the finite word n, and for an infinite word n we denote $|n| = \omega$. For a tree node n, the length $|n|$ is also the distance of n from the root of the tree (ε). A *branch* of the tree is a maximal sequence $n_0 n_1 n_2 \ldots$ (which can be either finite or infinite), such that $n_0 = \varepsilon$, and for all $i \geq 0$, $n_{i+1} \in children(n_i)$. If $t = (N, \ell)$ is a finite tree, the *front* of t is defined by $\text{front}(t) = \{n \in N \mid children(n) = \emptyset\}$, and the *height* of t is the length of the longest branch in t. Note that we measure height by the number of edges, not the number of nodes.

A *run* (or *run-tree*) of an automaton A on an infinite word $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega$ is a Q-tree $r = (N, \ell)$, such that for all $n \in N$, $children(n) \models \delta(\ell(n), w_{|n|})$. We say that a run r is *accepting* if for every branch $n_0 n_1 n_2 \ldots$ of r, some accepting state $q \in F$ appears infinitely often on the branch (that is, $\ell(n_i) = q$ for infinitely many values of i). If A has some accepting run on a word w, we say that A *accepts* w. The language of A, denoted $\mathcal{L}(A)$, is the set of words $w \in \Sigma^\omega$ such that A accepts w. Note that runs can be finite or infinite trees, and even in an infinite run there can be finite branches. However, finite branches must always end in a node n such that $\delta(n, w_{|n|}) = \textbf{true}$.

To model programs, we use Kripke structures. Given a set AP of atomic propositions, a *Kripke structure* (or *model*) over AP is a tuple $M = (S, s_0, R, L)$, where S is the state-space of the model, $s_0 \in S$ is the initial state, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{AP}$ is a labeling function which assigns to each model state a set of atomic propositions from AP. A *path* of M is a maximal sequence $\pi = s_0 s_1 s_2 \ldots$ starting at s_0, such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. The *labeling* of a path $\pi = s_0 s_1 s_2 \ldots$ is the word $L(\pi) = L(s_0)L(s_1)L(s_2)\ldots$.

Two parameters are often used to measure the complexity of a Kripke structure. The *diameter* d_M of a structure M is the length of the longest shortest path in M. The *recurrence diameter* r_M is the length of the longest loop-free path in M. The diameter of a model is no greater than its recurrence diameter,

since a shortest path is always loop-free, but the recurrence diameter is easier to compute than the diameter.

We say that a model $M = (S, s_0, R, L)$ *satisfies* an (existentially-interpreted) automaton $A = (2^{AP}, Q, q_0, \delta, F)$, and denote $M \models A$, if there is a path π of M such that $L(\pi) \in \mathcal{L}(A)$. The path π is then called a *witness*. In bounded model-checking (and linear-time model-checking in general), the automaton A describes the bad behaviors of the system, and a witness, if one exists, represents a bug in the system.

One way to check if $M \models A$ is to construct the *product* of M and A [20], an alternating automaton which, informally, describes all the runs that A can have on paths of M. The product automaton is defined by $M \times A = (\{a\}, S \times Q, (s_0, q_0), \delta_M, S \times F)$. The transition relation δ_M of the product automaton is defined by $\delta_M((s,q),a) = \bigvee_{s':(s,s')\in R} \alpha_{q,s,s'}$, where $\alpha_{q,s,s'}$ is the formula obtained from $\delta(q, L(s))$ by replacing every atom $q' \in Q$ with (s', q'). (For example, if $\delta(q, \{b\}) = q_1 \wedge q_2$, $L(s) = \{b\}$, and the only transitions from s are to s_1 and to s_2, then $\delta_M((s,q),a) = ((s_1,q_1) \wedge (s_1,q_2)) \vee ((s_2,q_1) \wedge (s_2,q_2))$.)

It can be shown that $M \models A$ iff $\mathcal{L}(M \times A) \neq \emptyset$. Therefore, to check if $M \models A$, we can construct $M \times A$ and check whether or not its language is empty [20]. Note that the product automaton $M \times A$ is over a *unary* alphabet $\{a\}$, and thus, if $\mathcal{L}(M \times A) \neq \emptyset$, then $M \times A$ must accept the word a^ω.

Throughout the paper we will be interested in prefixes of words and trees. We denote $x \prec y$ if x is a prefix of y. Given a finite or infinite word $x = x_0 x_1 \ldots$ and a number $h \leq |x|$, we denote $\mathrm{pref}_h(x) = x_0 \ldots x_h$. We use $\mathrm{pref}(x)$ (witout the subscript) to denote the set of prefixes of x, that is, $\mathrm{pref}(x) = \{\mathrm{pref}_h(x) \mid 0 \leq h \leq |x|\}$. Similarly, given a finite or infinite tree $t = (N, \ell)$, we use $\mathrm{pref}_h(t)$ to denote the tree $\mathrm{pref}_h(t) = (N_h, \ell_h)$ defined by $N_h = \{n \in N \mid |n| \leq h\}$ and $\ell_h(n) = \ell(n)$ for all $n \in N_h$. We also denote $\mathrm{pref}(t) = \{\mathrm{pref}_h(t) \mid h \in \mathbb{N}\}$. Finally, for an automaton A, we denote by $\mathrm{pref}_h(A) = \{\mathrm{pref}_h(r) | r \text{ is a run of } A\}$ the set of all prefixes of height h of runs of A. For a model M, we denote $\mathrm{pref}_h(M) = \{\mathrm{pref}_h(\pi) \mid \pi \text{ is a path of } M\}$.

4 A Dynamic Completeness Criterion for Alternating Automata

In this section we define a dynamic completeness criterion, which checks whether the automaton has a "beginning" of an accepting computation of length k. If there is no such "beginning", bounded model-checking can terminate and return $M \not\models A$ when the bound reaches k.

To formalize the notion of a "beginning" of an accepting computation, we define *canonical partial runs* of the automaton. Informally, a *partial run* is a truncated run-tree; it is a finite tree in which only the inner nodes are required to satisfy the transition relation. A *canonical* partial run is a partial run that contains no "useless" loops. We will later discuss another restriction on partial runs that may lead to smaller completeness thresholds.

Fix an alternating Büchi automaton on infinite words $A = (2^{AP}, Q, q_0, \delta, F)$.

Definition 1. *A* partial run of height h *of* A *on a word* $w = w_0 w_1 \ldots$ *is a* Q*-tree* $r = (N, \ell)$ *of height* h *satisfying the following conditions.*

1. $\ell(\varepsilon) = q_0$.
2. *For all* $n \in N$ *such that* $|n| < h$, *children*$(n) \models \delta(\ell(n), w_{|n|})$.

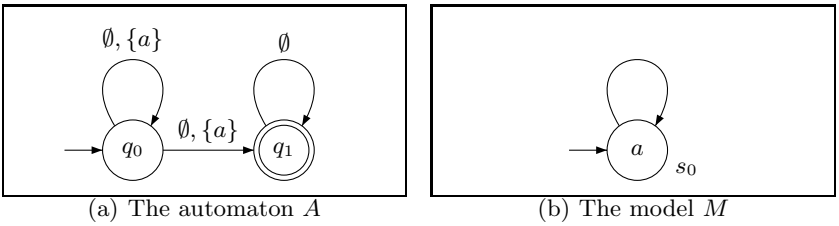
The definition of a partial run of height h is distinguished from the definition of an accepting run of the same height by the requirement on the leaves (nodes n with $|n| = h$): in an accepting run, if n is a leaf then $\delta(n, w_{|n|}) = \mathbf{true}$; in a partial run, there is no such requirement. Consequently, every accepting run is also a partial run, but the converse is not true.

Lemma 1. *If* r *is an accepting run on* w, *then for all* $h \in \mathbb{N}$, $\mathrm{pref}_h(r)$ *is a partial run of height* h *on* w.

Clearly, if A has an accepting run of height h or greater on a word w, then A has a partial run of height h' on w for all $h' \leq h$. However, there may exist partial runs of height h for all $h \in \mathbb{N}$ even if there is no accepting run, as in the following example.

Example 1. Consider the nondeterministic automaton A shown in Fig. 1(a). A accepts the language of words over the alphabet $\Sigma = \{\emptyset, \{a\}\}$ which contain a finite number of occurrences of the letter $\{a\}$. A run of A stays in q_0 until A "guesses" it has seen the last $\{a\}$, and then moves to q_1, which is an accepting state; if an $\{a\}$ is read from state q_1, the run gets stuck (that is, the transition is **false**).

The model M shown in Fig. 1(b) is not accepted by the automaton: the only path in M is $\pi = (s_0)^\omega$, whose labeling, $L(\pi) = (\{a\})^\omega$, contains infinitely many occurrences of $\{a\}$. However, for all $h \in \mathbb{N}$, the sequence $r_h = q_0^h$ represents a partial run of height h of A on π.



(a) The automaton A (b) The model M

Fig. 1. The automaton and model from Example 1

Notice that for $h > 2$, r_h is not a "good" partial run on π: after one step we enter a loop in both r and π in which no accepting state of A appears. Thus, r_h for $h > 1$ contains unnecessary padding which increases its height. Intuitively, a "good" partial run will contain no unnecessary loops. We now formalize this notion.

Definition 2. *Given a partial run* $r = (N, \ell)$ *of height* h,

1. *A* loop *in* r *is a pair* $(n_1, n_2) \in N^2$ *such that* $n_1 \prec n_2$ *and* $\ell(n_1) = \ell(n_2)$.
2. *We say that an automaton state* $q \in Q$ occurs in (n_1, n_2) *if there is a node* $n \in N$ *such that* $n_1 \prec n \prec n_2$ *and* $\ell(n) = q$.

3. A loop (n_1, n_2) *is said to be* useless *if no state* $q \in F$ *occurs in* (n_1, n_2).
4. We say that r *is* canonical *if there are no useless loops in* r.

Lemma 2. *If an automaton* A *over a unary alphabet* $\{a\}$ *accepts the word* a^ω, *then* A *has a canonical accepting run on* a^ω.

Proof sketch. The existence of a canonical accepting run follows from the existence of a memoryless winning strategy, shown in [9] for alternating parity automata, a more general class of automata than alternating Büchi automata. The accepting run that corresponds to a memoryless winning strategy for the word a^ω has no useless loops: if it had a useless loop, then the run-tree would contain a branch along which the loop repeats forever, because each time the automaton would reach each state in the loop it would be obliged by the memoryless strategy to make the same move. Since the loop contains no accepting state, this branch would contain only finitely many accepting states, and the run would not be accepting. □

The definition of a canonical partial run captures the notion of a "beginning" of a counter-example used in the dynamic completeness criteria of, e.g., [2], [22], [24]. The threshold we suggest is as follows.

Definition 3. *Given an automaton* A *and a model* M, *the* dynamic completeness threshold $\mathrm{CT}(M, A)$ *is the minimal number* h *such that* $M \times A$ *does not have a canonical partial run of height* h *(on* a^ω*), or* ∞ *if there is no such number.*

Next we show that the dynamic completeness threshold is sound — that is, it does not cause termination too early.

Theorem 1. *If* $M \models A$, *but for all* $h' < h$ *the product automaton* $M \times A$ *has no accepting run of height* h', *then* $\mathrm{CT}(M, A) \geq h$.

Proof. From Lemma 2, since $M \models A$, the product automaton $M \times A$ has a canonical accepting run r on a^ω. The run r is of height at least h, because $M \times A$ has no accepting runs of height smaller than h. Let $r' = \mathrm{pref}_h(r)$. From Lemma 1, r' is a partial run, and since r has no useless loops, neither does r'. Thus, r' is a canonical partial run of height h of $M \times A$, and $\mathrm{CT}(M, A) > h$.

Remark 1. Consider the case of a nondeterministic automaton A. A canonical partial run of A is a sequence of states — that is, a tree with a single branch. If the run contains a loop, then an accepting state must appear in the loop, implying the existence of an infinite accepting run of A.

Thus, for a nondeterministic automaton, the existence of a canonical partial run of height k indicates the existence of either an accepting run or a loop-free run of height k. However, the dynamic completeness criterion is only applied after we *fail* to find an accepting run at the current bound k; thus, we can rule out the first case, and simply search for a loop-free run of length k. This yields the length of the longest loop-free path in the product automaton as an upper bound on the completeness threshold of nondeterministic automata, as already observed in [6]. This is also the basis for the forward-traversal termination criteria of [2].

Table 1. The transition of the automaton from Example 2

q	$\delta(q,a)$	$\delta(q,b)$	$\delta(q,c)$
q_0	$q_0 \wedge q_a$	$q_0 \wedge q_b$	q_0
q_a	q_a	q_a	**true**
q_b	q_b	q_b	**false**

4.1 Eliminating Bad Prefixes

As a consequence of Theorem 1, if no accepting run of $M \times A$ has been found when the bound reaches $\mathrm{CT}(M,A)$, then bounded model-checking can halt and return $M \not\models A$. Conversely, we would expect that while $k < \mathrm{CT}(M,A)$ — that is, if there exists a canonical partial run of height k — then there should be "a reason" to increase the bound and search for an accepting run of height greater than k. However, the following example shows that this is not always the case.

Example 2. Consider the automaton $A = (\{a,b,c\}, \{q_0,q_a,q_b\}, q_0, \delta, \{q_0,q_b\})$, where the transition relation δ is given in Table 1. A accepts only words that do not contain both an a and a b: when an a is read, A moves to state q_a, where it waits to read a c; and when a b is read, A moves to state q_b, in which it must *not* read a c. Thus, any word in which both an a and a b appear will not be accepted. However, A still has a canonical partial run r of height 1 on any word w which has $\mathrm{pref}_1(w) = ab$. The partial run r cannot be extended into an accepting run regardless of the rest of w, but its existence may cause BMC not to terminate with a bound of 1 if the termination criterion from Definition 3 is used.

A prefix which cannot be extended into a word in the language, like ab in Example 2, is called a *bad prefix* [8]. One way to avoid bad prefixes is to construct a *prefix automaton* A_{pref} for A — an automaton which accepts a finite word iff it can be extended into an infinite word in the language of A — and to use it in the completeness criterion. This option is suitable for criteria based on nondeterministic automata (e.g., [2]), where a prefix automaton is constructed by simply removing any states from which there is no accepting run on any word and making all remaining states accepting. For alternating automata, however, this option is not suitable: [1] shows that for alternating automata the size of a prefix automaton can be exponential in the size of the original automaton. Next we present an alternative, which does not require the use of the prefix automaton in the dynamic criterion.

Definition 4. *Given an automaton $A = (\Sigma, Q, q_0, \delta, F)$, a set of automaton states $Q' \subseteq Q$ is said to be* A-consistent *if $\bigcap_{q \in Q'} \mathcal{L}(A_q) \neq \emptyset$, where the automaton A_q is defined by $A_q = (\Sigma, Q, q, \delta, F)$ (with q replacing q_0 as initial state).*

Now we augment the definition of canonical partial runs as follows.

Definition 5. *A partial run r of $M \times A$ is said to be* prefix-canonical *if r is canonical and the set $R = \{q \in Q \mid$ there exists $s \in S$ such that $(s,q) \in \mathrm{front}(r)\}$ is A-consistent.*

To compute the sets of A-consistent (or A-inconsistent) sets in an automaton A, we can build an equivalent nondeterministic automaton A' using the Miyano-Hayashi construction [16], which is a modified subset construction, and check which states of A' have some accepting computation. We do not go into the details of the construction for lack of space. The construction is exponential in the size of A, but it involves only the automaton and not the model. In addition, identifying the A-consistent sets is a preprocessing step which only needs to be performed once per automaton, and does not need to be repeated in every iteration of BMC.

The completeness threshold can be strengthened by adding the new requirement, and defining the *prefix completeness threshold* $\mathrm{CT}_P(M, A)$ to be the smallest number h such that $M \times A$ does not have a prefix-canonical partial run of height h. It is easy to show the equivalent of Theorem 1 for $\mathrm{CT}_P(M, A)$. In addition, we can now also show the following lemma and corollary, which give us a a reason why bounded model-checking should continue if the threshold has not been reached.

Lemma 3. *If A has a prefix-canonical partial run on a finite word $w \in \Sigma^h$, then there exists an infinite word $w' \in \mathcal{L}(A)$ such that $\mathrm{pref}_h(w') = w$.*

Corollary 1. *If $M \not\models A$ and $\mathrm{CT}_P(M, A) \geq h$, then there exists a model M' such that $\mathrm{pref}_h(M \times A) \subseteq \mathrm{pref}_h(M' \times A)$ and $M' \models A$.*

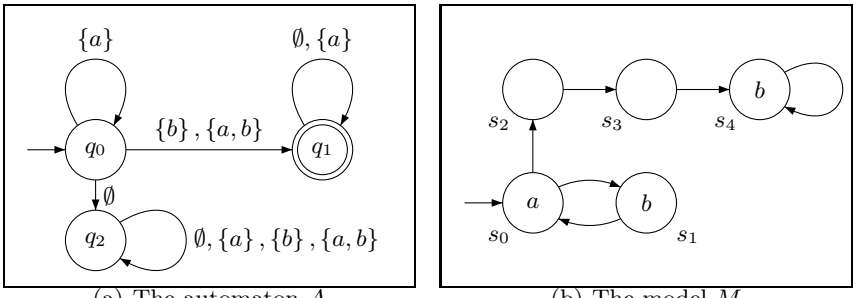
Corollary 1 shows that as long as the threshold has not been reached, there is a model M' which does satisfy A, such that any computation of $M \times A$ is also a computation of $M' \times A$. Informally, the fragment of M that A "can see" to depth h also exists in M', but $M' \models A$, and so we cannot stop searching for a witness just yet.

4.2 The Limitations of Existential Dynamic Completeness Criteria

It might seem better to require, instead of Corollary 1, that if $\mathrm{CT}_P(M, A) \geq h$ then there should exist a model M' such that $\mathrm{pref}_h(M \times A) = \mathrm{pref}_h(M' \times A)$ (equality instead of containment) and $M' \models A$. This stronger requirement, if satisfied, would imply that the completeness threshold uses all the information that is available in $\mathrm{pref}_h(M)$: if the threshold has not been reached, then there is a model which is *indistinguishable* from M to a depth of h as far as A is concerned, which A accepts. However, the following example shows that no sound completeness criterion of the form "there exists a path $\pi \in \mathrm{pref}_h(M)$ which satisfies ψ", where ψ is some condition on paths of length h, can satisfy the stronger requirement.

Example 3. Consider the model M shown in Fig. 2(b) , and the nondeterministic automaton A shown in Fig. 2(a). A accepts paths in which there is exactly one state labeled with b, and all the states before that state are labeled with a. $M \not\models A$, since all infinite paths of M contain at least two states labeled with b.

Suppose $\mathcal{C}$ is a dynamic completeness threshold of the form: "$\mathcal{C}(M, A)$ is the smallest number h such that there does not exist a path $\pi \in \mathrm{pref}_h(M)$ satisfying the condition ψ", where ψ is some condition on paths of length h. Suppose by way of contradiction that $\mathcal{C}$ also satisfies the following two conditions:



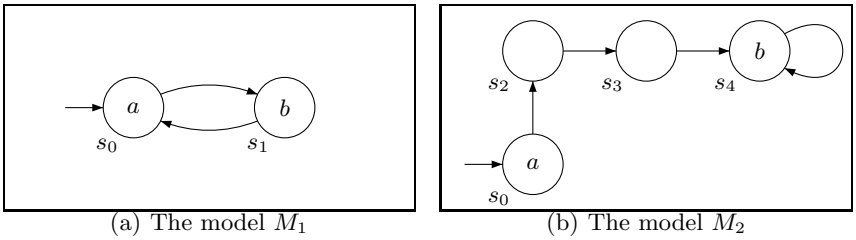
(a) The automaton A (b) The model M

Fig. 2. The automaton and model from Example 3

1. (Soundness) If $M \models A$, but $M \times A$ has no accepting run of height $h' < h$, then $\mathcal{C}(M, A) \geq h$,
2. If $\mathcal{C}(M, A) \geq h$ then there exists a model M' such that $\text{pref}_h(M \times A) = \text{pref}_h(M' \times A)$ and $M' \models A$.

In our case, $M \not\models A$. However, for $h = 2$, there exists a model M', obtained from M by changing the labeling of s_4 to $L'(s_4) = \emptyset$, such that $\text{pref}_h(M) = \text{pref}_h(M')$ and $M' \models A$. Any accepting run of $M' \times A$ must be infinite (since A has no **true** transitions). Therefore, for M', A, and $h = 2$, the terms of condition 1 are satisfied, and hence $\mathcal{C}(M', A) \geq 2$. From our assumption about the structure of $\mathcal{C}$, there must exist a path $\pi \in \text{pref}_2(M') = \text{pref}_2(M) = \{s_0 s_1 s_0, s_0 s_2 s_3\}$ which satisfies the condition ψ. We show that this contradicts condition 2.

If $\pi_1 = s_0 s_1 s_0$ satisfies the condition ψ, then the model M_1 shown in Fig. 3(a) also has $\mathcal{C}(M_1, A) \geq 2$, since $\pi_1 \in \text{pref}_2(M_1)$. Similarly, if $\pi_2 = s_0 s_2 s_3$ satisfies ψ, then the model M_2 shown in Fig. 3(b) has $\mathcal{C}(M_2, A) \geq 2$. However, for both $i = 1, 2$, there does not exist a model M_i' such that $\text{pref}_2(M_i \times A) = \text{pref}_2(M_i' \times A)$ and $M_i' \models A$. This is because in both cases, any attempt to extend M_i into a model satisfying A must add a transition from either s_0 (for M_1 or M_2) or from s_1 (for M_1), which will create a new path of length 2 in M and a new partial run in $\text{pref}_2(M_i' \times A)$. We thus have that any model M_i' such that $M_i' \models A$ and $\text{pref}_2(M_i \times A) \subseteq \text{pref}_2(M_i' \times A)$ must also have $\text{pref}_2(M_i') \not\subseteq \text{pref}_2(M_i)$, contradicting condition 2.



(a) The model M_1 (b) The model M_2

Fig. 3. The models M_1 and M_2 from Example 3

It is easy to extend Example 3 to conditions which talk about the existence of a path of arbitrary (but predetermined) length satisfying some condition ψ, not just paths of length h. The limitation we have shown is inherent to dynamic completeness thresholds based on existentially-interpreted automata.

5 The Gap between the Completeness Threshold for Alternating and Non-deterministic Automata

It is well known that although alternating automata are equivalent in expression power to nondeterministic automata, they are exponentially more succint: there exist languages recognized by an alternating automaton comprising n states, which cannot be recognized by a nondeterministic automaton that has less than 2^n states [20]. In this section we show that the ratio between the completeness thresholds for alternating and nondeterministic automata can also be exponential in the number of states of the alternating automaton.

In [16] it is shown that for any alternating automaton A with n states, there exists an equivalent nondeterministic automaton A' with $n' = 2^{2n}$ states, such that $\mathcal{L}(A) = \mathcal{L}(A')$. The following upper bound result follows in a straightforward manner, because an exponential blow-up in the number of states implies at worst an exponential blow-up in the size of loops in the run-tree.

Theorem 2. *For any alternating automaton A with n states and model M with diameter d, there is a nondeterministic automaton A' with $\mathcal{L}(A) = \mathcal{L}(A')$ such that* $\mathrm{CT}(A') \leq 2^{2n} \cdot d \cdot \mathrm{CT}(A)$.

In addition to the upper bound on the number of states, [20] shows a matching exponential lower-bound: there exists an alternating automaton with n states such that any equivalent nondeterministic automaton must have at least 2^n states. However, this does not immediately imply a corresponding lower bound on the completeness thresholds nondeterministic automata, because a large number of states does not necessarily translate to long loops in the runs of the automaton. For example, the family of languages used in [20] to show the exponential lower bound can be recognized by altnernating and nondeterministic automata that have the same completeness threshold on any given model (even though the nondeterministic automaton would require exponentially more states than the alternating automaton).

The lower bound we will show stems from the fact that in an alternating automaton we require that *each branch* of the run-tree not contain unnecessary loops. Thus, we can detect "hopeless situations" as soon as one branch cannot be extended without creating an unnecessary loop, even if other branches in the run-tree are loop-free. In contrast, the runs of a nondeterministic automaton only have one branch, and we show that this branch can be made to grow exponentially long without closing a loop. Our result is related to the ability of alternating automata to count to 2^n using only n states, while nondeterministic automata require 2^n states to accomplish the same task [13].

Theorem 3. *There is a model M over a single atomic proposition $AP = \{p\}$, and a family of languages $\{L_n\}_{n=1}^{\infty}$ over 2^{AP}, such that for all $n \geq 1$,*

1. *There is an alternating automaton A with $O(n)$ states such that $\mathcal{L}(A) = L_n$ and $\mathrm{CT}(M, A) = 1$, and*
2. *Any nondeterministic automaton A' such that $\mathcal{L}(A') = L_n$ must have $\mathrm{CT}(M, A') \geq 2^n - 1$.*

Proof. We will use the model $M = (\{s\}, s, \{(s, s)\}, L)$, where $L(s) = \emptyset$. For simplicity we will represent $2^{AP} = \{\emptyset, \{p\}\}$ by the binary alphabet $\Sigma = \{a, b\}$, where a stands for $\emptyset$ and b stands for $\{p\}$. We now construct the family $\{L_n\}_{n=1}^{\infty}$.

Let $n \geq 1$. Consider the finitary language $H_n = \left\{bw \mid w \in \Sigma^{2^n-1}\right\}$, which contains words of length 2^n that start with b. H_n can be recognized by a nondeterministic automaton with 2^n states. Consequently, by a result from [15], there exists an alternating automaton $A_1 = (\{a, b\}, Q_1, q_0^1, \delta_1, F_1)$ with n states which recognizes the reverse language $H_n^R = \left\{wb \mid w \in \Sigma^{2^n-1}\right\}$.

Let $G_n = \{uav \mid u \in \Sigma^*, v \in \Sigma^\omega\}$, and let $L_n = (H_n^R \cdot \Sigma^\omega) \cap G_n$. L_n is the language of infinite words in which the letter b appears in position 2^n and the letter a appears anywhere in the word. L_n can be recognized by an alternating automaton A_L, defined by $A_L = (\{a, b\}, Q_1 \cup \{q_0, q_a\}, q_0, \delta_L, F_1)$, where we assume w.l.o.g. that $q_0, q_a \notin Q_1$, and δ_L is defined as follows:

- For all $q \in Q_1$ and $\sigma \in \{a, b\}$, $\delta(q, \sigma) = \delta_1(q, \sigma)$. (This part of the automaton behaves exactly like A_1.)
- $\delta(q_0, a) = \delta_1(q_0^1, a)$. (When reading an a from the initial state, the obligation that a must eventually be seen is discharged immediately, and the automaton behaves like A_1 from that point onward.)
- $\delta(q_0, b) = q_a \wedge \delta_1(q_0^1, b)$. (When reading a b from the initial state, the automaton splits into two parts, one that behaves like A_1 and one that waits to see an a.)
- $\delta(q_a, a) = \mathbf{true}$ and $\delta(q_a, b) = q_a$. (In q_a, the automaton waits to see an a. Note that since $q_a \notin F_1$, a branch along which only q_a appears is not accepting. Thus, an a must appear eventually in order for the word to be accepted.)

The automaton A_L has $n + 2$ states. Let r be a partial run of height h of $M \times A$. The only path in M is $\pi = s^\omega$, labeled $L(\pi) = b^\omega$. From the definition of δ_L, the run r contains a branch of the form $(s, q_0)(s, q_a)^h$. But q_a is not an accepting state in A_L, and therefore (s, q_a) is not accepting in $M \times A_L$. Thus, if $h > 1$, then the branch $(s, q_0)(s, q_a)^h$ contains an unnecessary loop, and r is not canonical. Consequently we have that $M \times A_L$ has no canonical partial run of length 2 or greater, and $\mathrm{CT}(M, A_L) = 1$.

Now suppose that A' is a nondeterministic automaton which recognizes L_n. Then A' accepts the word $w = b^{2^n} a^\omega \in L_n$. Let $t = q_0 q_1 \ldots$ be an accepting run of A' on w, and consider the path $\pi = s^\omega$ in M. Note that $\mathrm{pref}_{2^n-1}(L(\pi)) = \mathrm{pref}_{2^n-1}(w) = b^{2^n}$, and hence it is easy to verify that the sequence $t_M = (s, q_0)(s, q_1) \ldots (s, q_{2^n})$ is a partial run of height $2^n - 1$ of $M \times A'$.

Suppose by way of contradiction that $\mathrm{CT}(M, A') < 2^n - 1$, that is, $M \times A'$ has no canonical partial runs of height $2^n - 1$. Then the run t_M cannot be canonical, since its height is $2^n - 1$. We therefore have that t_M contains a useless loop; in particular, there exist $i < j \leq 2^n$ such that $q_i = q_j$. It is easily shown that the run

$t' = q_0 \dots q_{i-1} q_j q_{j+1} \dots$ is an accepting run of A' on the word $w' = b^{2^n-(j-i)}a^\omega$. But because $i < j$, the letter b does not appear in position 2^n of w', and $w' \notin L_n$. This contradicts our assumption that $\mathcal{L}(A') = L_n$. □

Note that the completeness threshold of the alternating automaton from Theorem 3 *remains constant* as n grows, while the completeness threshold of the corresponding nondeterministic automaton grows exponentially with n.

6 Conclusion

We developed a dynamic completeness criterion for bounded model-checking, which we believe explains and generalizes several dynamic completeness criteria that were developed for various encodings and temporal logics. By using automata as the specification mechanism we were able to abstract away the details of the specific temporal logic and encoding, and obtain a notion of a "beginning" of a bad behavior which is applicable to the full class of ω-regular properties.

We also showed that alternating automata are better suited to serve as a basis for completeness criteria than nondeterministic automata: alternating automata can "separate concerns" and track different requirements in different branches of the run-tree, which can lead to termination as soon as we verify that at least one of the requirements cannot be fulfilled. We are interested in developing an encoding for our completeness criterion based on the encoding from [11] for weak alternating Büchi automata.

Acknowledgements. The author is grateful to Orna Grumberg, Yoram Moses and Avi Yadgar for many fruitful discussions.

References

1. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.Y.: Resets vs. aborts in linear temporal logic. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 65–80. Springer, Heidelberg (2003)
2. Awedh, M., Somenzi, F.: Proving more properties with bounded model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 96–108. Springer, Heidelberg (2004)
3. Awedh, M., Somenzi, F.: Termination criteria for bounded model checking: Extensions and comparison. Electr. Notes Theor. Comput. Sci. 144(1), 51–66 (2006)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. 19(1), 7–34 (2001)
6. Clarke, E., Kroening, D., Strichman, O., Ouaknine, J.: Completeness and Complexity of Bounded Model Checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004)
7. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)

8. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
9. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: SFCS 1991, Washington, DC, USA, pp. 368–377. IEEE Computer Society, Los Alamitos (1991)
10. Heljanko, K., Junttila, T., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
11. Heljanko, K., Junttila, T.A., Keinänen, M., Lange, M., Latvala, T.: Bounded model checking for weak alternating büchi automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 95–108. Springer, Heidelberg (2006)
12. Jehle, M., Johannsen, J., Lange, M., Rachinsky, N.: Bounded model checking for all regular properties. In: BMC 2005. Electr. Notes in Theor. Comp. Sc., vol. 144, pp. 3–18. Elsevier, Amsterdam (2005)
13. Kupferman, O., Ta-shma, A., Vardi, M.Y.: Counting with automata. Technical report (1999)
14. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM 47(2), 312–360 (2000)
15. Leiss, E.L.: Succint representation of regular languages by boolean automata. Theor. Comput. Sci. 13, 323–330 (1981)
16. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. In: CAAP 1984, pp. 195–210 (1984)
17. Oshman, R.: Bounded model-checking for universal branching-time logic. Master's thesis, Technion – Israel Institute of Technology (August 2008)
18. Oshman, R., Grumberg, O.: A new approach to bounded model checking for branching time logics. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 410–424. Springer, Heidelberg (2007)
19. Penczek, W., Wozna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of CTL. Fundam. Inf. 51(1), 135–156 (2002)
20. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
21. Y. Vardi, M.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
22. Wang, B.-Y.: Proving forall-mu-calculus properties with SAT-based model checking. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 113–127. Springer, Heidelberg (2005)
23. Woźna, B.: Bounded Model Checking for the universal fragment of CTL*. Fundamenta Informaticae 63(1), 65–87 (2004)
24. Zhang, W.: Verification of actl properties by bounded model checking. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 556–563. Springer, Heidelberg (2007)

A Scalable Memory Model for Low-Level Code*

Zvonimir Rakamarić and Alan J. Hu

Department of Computer Science, University of British Columbia, Canada
{zrakamar,ajh}@cs.ubc.ca

Abstract. Because of its critical importance underlying all other software, low-level system software is among the most important targets for formal verification. Low-level systems software must sometimes make type-unsafe memory accesses, but because of the vast size of available heap memory in today's computer systems, faithfully representing each memory allocation and access does not scale when analyzing large programs. Instead, verification tools rely on abstract memory models to represent the program heap. This paper reports on two related investigations to develop an accurate (i.e., providing a useful level of soundness and precision) and scalable memory model: First, we compare a recently introduced memory model, specifically designed to more accurately model low-level memory accesses in systems code, to an older, widely adopted memory model. Unfortunately, we find that the newer memory model scales poorly compared to the earlier, less accurate model. Next, we investigate how to improve the soundness of the less accurate model. A direct approach is to add assertions to the code that each memory access does not break the assumptions of the memory model, but this causes verification complexity to blow-up. Instead, we develop a novel, extremely lightweight static analysis that quickly and conservatively guarantees that most memory accesses safely respect the assumptions of the memory model, thereby eliminating almost all of these extra type-checking assertions. Furthermore, this analysis allows us to create automatically memory models that flexibly use the more scalable memory model for most of memory, but resorting to a more accurate model for memory accesses that might need it.

1 Introduction

Because of its critical importance underlying all other software, low-level system software is among the most important targets for formal verification. For example, the correct execution of even the most mundane software relies on a vast array of supporting system software: the compiler and linker during development, of course, but also all the OS services at runtime: application-level memory management and the underlying virtual memory system, context swaps and the underlying OS scheduler, device drivers for all I/O, etc. With the emergence of virtualization, the hypervisor becomes an even lower-level, even more critical layer that needs verification (e.g., [27]), as even the operating system relies on its correctness.

All formal software analysis must model memory in some way. At one extreme, the entire memory space could be modeled as a single, giant array of bytes/words

* This work was supported by a research grant from the Natural Sciences and Engineering Research Council of Canada and a University of British Columbia Graduate Fellowship.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 290–304, 2009.

(e.g., [10, 11, 14], early versions of VCC [27] also supported byte-level reasoning). Doing so makes the verification completely accurate (sound and precise with respect to the effect of any memory access), but does not scale beyond very small segments of code. At the other extreme, we can restrict our analysis to handle only code that has no dynamic memory allocation and is completely type-safe (e.g., [6])[1]. Such an approach has scaled to millions of lines of code [6], but obviously precludes verification of typical mainstream software. Most software verification tools (e.g., [2, 8, 18, 20, 21]) try to strike a balance, assuming some degree of type-safety, e.g., assuming that pointers to different types of objects do not alias. Note that most tools do not check these assumptions — if the code violates the assumption, the tool might report wrong answers without any warning.

The choice of memory model is particularly challenging for low-level systems software, because such software must sometimes make type-unsafe memory accesses. For example, common idioms include casting a data structure from/into an array of bytes or integers for efficiency or to interface to hardware, and accessing a structure via differently-typed pointers as a way to implement sub-typing in C. Address arithmetic is also common, usually to offset before or after a given pointer in order to access a nearby data field. Verification tools for low-level software must find an intermediate memory model that assumes some type information to provide scalability, yet accurately captures the effects of lower-level, type-unsafe memory accesses.

In this paper, we develop such a model. The paper consists of two separate, but related parts. In the first part (Section 2), we compare a recently introduced memory model, specifically designed to more accurately model low-level memory accesses in systems code, to an older, widely adopted memory model. We find that the newer memory model scales poorly compared to the earlier, less accurate model. In the second part (Section 3), we investigate how to improve the soundness of the less accurate model. We first consider adding assertions to the code that each memory access does not break the assumptions of the memory model, but this causes verification complexity to blow-up. Then, we develop a novel, extremely lightweight static analysis that quickly and conservatively guarantees that most memory accesses safely respect the assumptions of the memory model, thereby eliminating almost all of these extra type-checking assertions. Furthermore, this analysis allows us to create automatically memory models that flexibly use the more scalable memory model for most of memory, but resort to a more accurate model for memory accesses that might need it. Experimental results show that the static analysis is very fast, maintaining the scalability of the less accurate memory model. Along the way, our tool found four bugs in real Linux device drivers, three of which were previously unreported.

2 Comparing Two Memory Models

Because of the vast size of available memory in today's computer systems, faithfully representing each memory allocation and access in a static verifier does not scale. Therefore, verification tools rely on memory models that trade precision for scalability, and in turn, they define programming language operational semantics with respect to

[1] Astrée now supports type casts, but still does not support dynamically allocated memory [24].

the chosen memory model. In this section, we introduce two memory models that are typically used in modular deductive verification tools, describe their advantages and drawbacks in the context of low-level code verification, and present empirical results on using the models to verify a number of Linux device drivers.

2.1 Monolithic Memory Model

Our first memory model is heavily influenced by the one used in early versions of HAVOC [9], and also similar to the one used in the first incarnation of VCC [27]. The main idea behind this memory model is to divide the memory into disjoint objects (or regions). Each object is identified by its reference, and has a fixed size determined when the object is allocated. A pointer in the memory model is therefore a pair, consisting of a reference and an offset; the reference uniquely defines the object into which the pointer points; the byte offset defines the byte in the object being pointed to.

To be able to translate a program into a representation that uses a memory model, we have to define the semantics of its source language with respect to the chosen memory model. In the monolithic memory model, the semantics of programs depends on three fundamental types: the uninterpreted type `ref` of object references, the type `int` of integers, and the type $\texttt{ptr} = \texttt{ref} \times \texttt{int}$ of pointers. For notational convenience, each variable in a program, regardless of its declared type, contains a pointer value: a pointer is a pair containing an object reference and an integer offset, and an integer value is encoded as a pointer value whose first component is the special constant `null` of type `ref`. Note that because of the integer offset component, the memory model can precisely capture byte offsets and low-level pointer arithmetic inside an object. On the other hand, since object references are uninterpreted, the objects are essentially "infinitely apart", and the memory model cannot model pointer arithmetic between objects.

The heap of a program is modeled using two map variables, `Mem` and `Alloc`, and a map constant `Size`:

$$\begin{aligned} \texttt{Mem} &: \texttt{ptr} \rightarrow \texttt{ptr} \\ \texttt{Alloc} &: \texttt{ref} \rightarrow \{\texttt{UNALLOCATED}, \texttt{ALLOCATED}\} \\ \texttt{Size} &: \texttt{ref} \rightarrow \texttt{int} \end{aligned}$$

The variable `Mem` maps pointers to pointers and represents the contents of memory at a location. The variable `Alloc` maps object references to the set {`UNALLOCATED`, `ALLOCATED`} and is used to model memory allocation. The constant `Size` maps object references to positive integers and represents the size of the object. For instance, the procedure call `malloc(n)` for allocating a memory buffer of size `n` returns a pointer $\texttt{Ptr}(o,0)$ where o is an object reference such that $\texttt{Alloc}[o] = \texttt{UNALLOCATED}$ and $\texttt{Size}[o] \geq n$ before the call, and $\texttt{Alloc}[o] = \texttt{ALLOCATED}$ after the call (ignoring the possibility of memory allocation failure, which could also be easily modeled).

2.2 Burstall's Memory Model

Our second memory model is a type-indexed memory model (also known as Burstall's memory model [7]) that has been commonly used in the deductive verification of type-safe languages [5, 19]. The main idea behind this model is that, apart from dividing

memory into disjoint objects as in the previous model, we also split the memory according to a set of possible *types* of memory locations. To achieve this splitting, a set of unique type constants of type `type` is introduced, which represent types in the original program. The common types found in a language, such as `int`, `int*`, `char`, etc., are going to be translated as type constants `$int`, `$intP`, `$char`, etc. Usually, apart from all of the commonly found types, the set of type constants also contains a unique type constant for each structure field. For instance, the structure

```
struct {
  int x;
  int y;
} foo;
```

introduces unique type constants `$foo#x` and `$foo#y`. It turns out that this "type-awareness" in the model, caused by adding type constants and splitting the memory according to those, is exactly what gives this model an edge when it comes to scalability over the monolithic model.

Our map `Mem` from the previous memory model is therefore, instead of mapping pointers to pointers, going to map type-pointer pairs to pointers. We also introduce in the model an additional map constant `Type` that maps pointers (memory locations) to types and represents the allocation type of memory locations. Each type in the memory model is a unique constant distinct from all other types. The type-indexed memory model therefore has four maps:

$$\begin{aligned} \texttt{Mem} &: (\texttt{type} \times \texttt{ptr}) \to \texttt{ptr} \\ \texttt{Alloc} &: \texttt{ref} \to \{\texttt{UNALLOCATED}, \texttt{ALLOCATED}\} \\ \texttt{Size} &: \texttt{ref} \to \texttt{int} \\ \texttt{Type} &: \texttt{ptr} \to \texttt{type} \end{aligned}$$

Adding types to the memory model makes proving programs easier and faster:

- One can conclude that updates to different fields of a structure don't influence each other without reasoning about integer offsets and pointer arithmetic, as would be needed in the monolithic memory model. Such reasoning is often hard in the presence of quantifiers.
- Memory locations of different fields of two distinct objects usually don't alias, which is nicely captured by this memory model. This also greatly simplifies the task of proving many interesting assertions.
- When a field is being updated, based on its type, only the corresponding submap of `Mem` changes, which simplifies proving frame axioms.

2.3 Experimental Results

We have implemented the preceding memory models as part of our tool SMACK (Static Modular Assertion ChecKer [26]), which is a modular, annotation-based, extended static property checker of C programs. In the spirit of modular verification, SMACK verifies programs annotated with procedure specifications and loop invariants. It uses the

Table 1. Running times for checking correct locking behavior in device drivers from the Linux kernel. The column "LOC" given the number of lines of code; "Monolithic" gives the total running time of BOOGIE using the monolithic memory model; "Burstall" gives the total running time of BOOGIE using Burstall's memory model with assumed types; "Speedup" compares the running times. The * indicates that BOOGIE timed out on two procedures from the applicom driver (time out is set to 1200s).

Driver	LOC	Memory Model		Speedup
		Monolithic (s)	Burstall (s)	
ib700wd	346	45.7	14.6	3.1
w83877f_wdt	421	59.5	16.1	3.7
sc520_wdt	443	50.2	16.5	3.0
machzwd	494	71.0	18.1	3.9
wdt977	519	46.4	19.3	2.4
ds1286	633	70.8	20.3	3.5
efirtc	815	62.2	16.3	3.8
applicom	934	*3368.8	161.2	20.9

LLVM compiler framework [22] to parse input programs and annotations. The LLVM output is translated by SMACK into a BoogiePL [16] program based on the operational semantics of C memory accesses according to the selected memory model. BoogiePL is the input language of the BOOGIE verifier [3], which, in turn, generates a verification condition (VC) from the input program whose validity implies partial correctness of the input. The VC generation in BOOGIE is performed using a variation [4] of the standard *weakest precondition* transformer [17]. We check the generated VC using the accompanying Z3 theorem prover [15]. We report only the running times of BOOGIE required to verify the examples since the transformation SMACK performs takes only a small fraction of that time.

We applied SMACK to check correct locking behavior of several device drivers from the Linux kernel. The source code of the examples, the models and stubs of the relevant kernel routines, and the test harness are taken from the DDVERIFY suite [1, 29]. Ensuring correct locking behavior amounts to checking that locks are initialized before they are used and that locks are alternately acquired and released. Table 1 lists the drivers and gives the running times for the verification using the monolithic and Burstall's memory models. All experiments were executed on an Intel Pentium D at 2.8Ghz.

Seven of the drivers were arbitrarily picked character device drivers that contain spinlocks, usually as one or two global variables. In addition, we handpicked the applicom driver, since this driver has a global array of structures where each structure is protected by its own spinlock. This makes it much more interesting and challenging to verify (see Figure 1), requiring from a tool the ability to reason precisely about such unbounded data structures. Current tools that are typically used in the verification of device drivers [2, 8, 10, 11, 20, 21] have trouble handling unbounded data structures. One of the goals of SMACK is to address that weakness.

From the running times, it can be seen that Burstall's memory model is the clear winner. It always outperforms the monolithic memory model on easier examples, and

```
1 struct applicom_board {
2   unsigned long PhysIO;
3   void __iomem *RamIO;
4   wait_queue_head_t FlagSleepSend;
5   long irq;
6   spinlock_t mutex;
7 } apbs[MAX_BOARD];
8
9 irqreturn_t ac_interrupt(int vec, void *dev_instance) {
10  for (i = 0; i < MAX_BOARD; i++) {
11    if (!apbs[i].RamIO) continue;
12    spin_lock(&apbs[i].mutex);
13    if(readb(apbs[i].RamIO + RAM_IT_TO_PC)) {
14      spin_unlock(&apbs[i].mutex);
15      i--;
16    } else {
17      spin_unlock(&apbs[i].mutex);
18    }
19  }
```

Fig. 1. Simplified code excerpt from the applicom Linux device driver illustrating the complexity of checking correct locking behavior. The loop on line 10 iterates over array elements. If the field `RamIO` of the element at index `i` is not null (line 11), the lock (field `mutex`) is acquired on line 12 and then later released. The verification requires checking complex invariants over all elements of the array (i.e. quantified) that involve values of the `RamIO` fields as well as the status of locks (initialized, locked, unlocked).

the speedup factor is from 2.4 to 3.9. Furthermore, using Burstall's memory model, we managed to verify the applicom example, which we couldn't do using the monolithic memory model since it timed out on two procedures. The example requires proving complex quantified invariants over fields from an array of structures. The key to successful verification of this example is structure field disambiguation: Burstall's memory model provides this for free, whereas in the monolithic model, it requires reasoning about offsets and pointer arithmetic.

However, the much better running times of Burstall's memory model come at a price: it relies on the assumption that memory is strongly typed. In the examples, when we use Burstall's model, we are assuming the type of a memory location before each memory access, which is unsound and can cause bugs to be missed in a type-unsafe setting such as C. In the next section, we describe how to deal with this problem.

3 Ensuring Soundness with Burstall's Memory Model

Burstall's memory model relies on the assumption that memory is strongly typed, as in type-safe languages such as Java. That means that a type of the object is established when it is created, via a call to `new`, and the object is always accessed using that original type. However, low-level languages like C allow reinterpretation of the original type and

```
1 typedef struct {
2   int x;
3 } S1;
4
5 typedef struct {
6   int a;
7   int b;
8 } S2;
9
10 void main() {
11   S2* s2 =
12     (S2*)malloc(sizeof(S2));
13   S1* s1 = (S1*)s2;
14
15   s2->a = 3;
16   s1->x = 4;
17
18   assert(s2->a == 3);
19 }
```

```
1 const unique $S1#x:type;
2 const unique $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   s1 := s2;
9
10  Mem[$S2#a,s2] := Ptr(null,3);
11  Mem[$S1#x,s1] := Ptr(null,4);
12
13  assert(Mem[$S2#a,s2] ==
14    Ptr(null,3));
15 }
```

Fig. 2. Example illustrating a simple upcasting in C that causes unsoundness in Burstall's memory model. The right column shows simplified BoogiePL code of the translation of the function `main`, assuming Burstall's model. Because of the assumption of type safety, the two assignments on BoogiePL lines 10 and 11 do not alias, resulting in the assertion incorrectly passing.

therefore type-unsafe memory accesses. Such operations are not uncommon in systems code and are typically done in C using casts or unions[2]. Often, casts don't reinterpret memory at the byte level, but are used to simulate object-oriented language features, such as inheritance, that are not supported directly in C. In fact, according to empirical studies [13, 28], more than 90% of the structure casts in C fall into that category.

Figure 2 gives a simple example illustrating "upcasting" in C. The structure `S2` is a subtype of the structure `S1`, and the cast on line 13 represents an upcast. The example shows how such a simple cast can cause Burstall's memory model to become unsound: the field update on line 16 overwrites the value that was written to the same memory location on line 15, and the assertion on line 18 fails. However, in Burstall's model this overwrite does not happen, since different field names (i.e. different unique types) denote different memory locations in the model: the write to `s2->a` is translated as the write to `Mem[$S2#a,s2]` on line 10 of the BoogiePL translation in the right column, while the write to `s1->x` is translated at the write to `Mem[$S1#x,s1]` on line 11, and doesn't overwrite the location `Mem[$S2#a,s2]` although the pointers `s1` and `s2` are equal.

A simple way of ensuring soundness in the presence of such casts is to syntactically analyze the source code and just give up on the verification if we find one (e.g., [18]). Our goal is to go a step further and verify the code even in the presence of type-unsafe structure casts, while preserving soundness. In the following sections, we'll describe three different techniques of how to achieve that goal.

[2] We can consider union a special case of cast.

```
1 const unique $S1#x:type;
2 const unique $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
9   s1 := s2;
10
11  assert(Type[s2] == $S2#a);
12  Mem[$S2#a,s2] := Ptr(null,3);
13  assert(Type[s1] == $S1#x); // Fails!
14  Mem[$S1#x,s1] := Ptr(null,4);
15
16  assert(Type[s2] == $S2#a);
17  assert(Mem[$S2#a,s2] == Ptr(null,3));
18 }
```

Fig. 3. Translation of the example from Fig. 2 with type-check assertions added before each memory access (lines 11, 13, and 16). The type-check assertion on line 13 will fail, indicating a violation of the assumption of type safety.

3.1 Guarding Memory Accesses with Type Assertions

A straightforward way of preventing unsoundness described in the previous section from happening in Burstall's memory model is to add *type checks* before each memory access. The checks are added in the form of assertions on the Type map. Every access to a memory location x with type $\$t$ is going to be preceeded with the assertion $assert(\texttt{Type}(x) == \$t)$ that will have to be discharged.

Figure 3 shows the translation of the example in Figure 2 with the inserted type checks. The map Type represents the compile-time allocation type of memory locations, and therefore the correct allocation type has to be assumed on line 8 after the allocation. Then, type check assertions are inserted before each memory access (lines 11, 13, and 16). The type check assertion on line 13 clearly will fail: $\texttt{s1} = \texttt{s2}$, and the type of s2 is $S2#a, not $S1#x. Whenever a memory location is accessed through a type that is not the allocation type of the memory location, the added type check assertion will fail. This preserves the soundness of the verification in Burstall's model.

However, proving such type check assertions for each memory access in the program is a big overhead, as we'll show later on in the experiments in Section 3.4. Furthermore, discharging those assertions often requires adding more manual annotations to the code which poses an additional burden on the user. Both of these drawbacks are an unacceptable burden that is not justified since most parts of the code usually obey the type restrictions imposed by Burstall's memory model. Therefore, in the next section, we introduce a lightweight static analysis that eagerly removes most of the required type-check assertions by conservatively guaranteeing that those memory accesses safely respect the assumptions of the model.

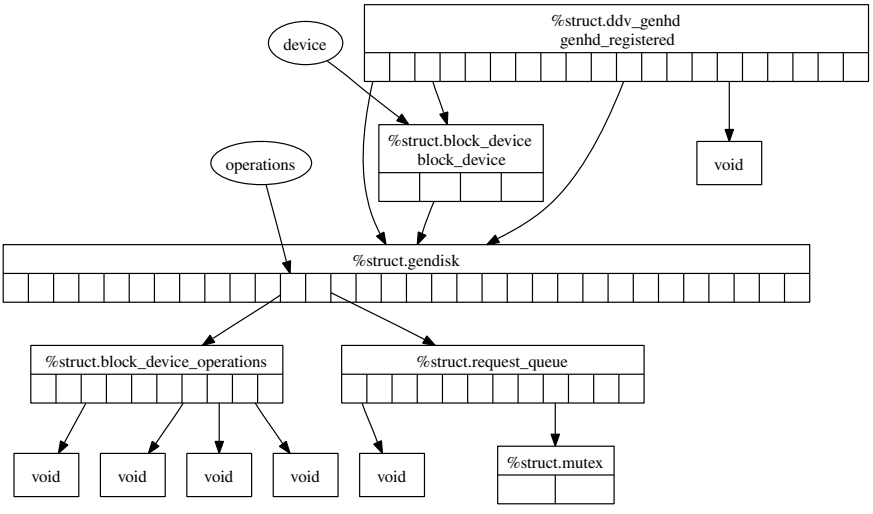


Fig. 4. An example of a Data Structure Graph. The graph shows a simplified part of the globals DS graph for the applicom device driver. Oval nodes in the graph are pointer variable nodes (e.g. *device* and *operations*); rectangle nodes are heap nodes (e.g. *genhd_registered* and *block_device*). Each heap node has a type. For instance, the type of the *genhd_registered* node is *struct.ddv_genhd*, the type of the *block_device* node is *struct.block_device*, etc. Pointer fields of heap nodes have outgoing edges, while fields of other types are just empty boxes.

3.2 Eagerly Eliminating Type Check Assertions

We'll start this section by giving some background information on the pointer analysis that is the starting point of our technique for eagerly eliminating type check assertions. Then, we'll describe our algorithm for eliminating type checks.

Data Structure Analysis (DSA). DSA [23] is a highly scalable and fast, context-sensitive (with full *heap cloning*), field-sensitive (even in a type-unsafe setting), conservative pointer analysis. The term "heap cloning" refers to a property important for achieving true context-sensitivity — heap objects are not distinguished just by allocation site, but also by (acyclic) call paths leading to their allocation, i.e. the calling context in which they were created. Support for data structure operations is often going to be encapsulated in a library used throughout the code, and therefore context-sensitivity is important to be able to handle such cases precisely.

DSA constructs a representation of the heap in the form of Data Structure Graphs (DS graphs); it creates one DS graph per procedure plus an additional one for global storage. The separate globals graph is a key optimization allowing procedure graphs to contain only the parts of global storage reachable from that procedure. A DS graph consists of a set of nodes (DS nodes) and a set of edges. As an example, a simplified part of the globals DS graph for the applicom device driver is shown in Figure 4. We distinguish two types of DS nodes: heap nodes with a number of fields at different offsets (e.g. rectangle nodes in the example graph), and pointer variable nodes that point into heap nodes (e.g. oval nodes in the example graph). A pointer variable node is named

after the pointer variable it represents and has one edge. A heap node has one outgoing edge per pointer field. Each heap node has a type and represents a potentially unbounded number of objects in memory of that type. A DS graph edge is defined by its source node and offset (i.e. offset of the respective pointer field in the source node), and its end node and offset. For instance, if the word size is 4 bytes, the second edge coming out of the *genhd_registered* node is defined by $\langle genhd\_registered, 8\rangle \rightarrow \langle block\_device, 0\rangle$.

Instead of just providing the usual pairs of references that may alias (points-to/alias information), the explicit heap representation DSA constructs can be used to identify different instances of data structures and provide structural and type information for each identified instance. The key feature of DSA we take advantage of is the conservative type information for each heap object. In particular, if all accesses to objects that a node represents obey a consistent type, such node is called "type-homogeneous". Accesses are defined as operations on pointers that point into the node and actually interpret the type: load and store operations, and structure and array indexing operations on pointers. Operations such as memory allocation and pointer casts (e.g. from `void*`) are not counted as accesses and don't influence a node type. If accesses with incompatible types are found, the type of the node is marked as *Unknown*. Therefore, DSA tracks types precisely in the type-safe parts of the heap/program, while in the presence of type-unsafe operations it conservatively treats nodes as having an unknown type.

Eager Type Check Elimination Algorithm. The algorithm is relatively simple and straightforward, but as we'll show in the experiments in Section 3.4, extremely effective. First, we run the DSA on the code we are analyzing, outputting a DS graph for each procedure and the globals graph. Then, for each memory read or write through a pointer, we find the type of the memory location it points to using the appropriate DS graph. If the computed type is the same as the actual type of the pointer, we omit the type check (assertion) that would be otherwise generated. If the types are not the same or if the type of the node the pointer points to is *Unknown*, we will generate the type check assertion to preserve soundness.

Figure 5 illustrates the benefits of our technique, removing two type-check assertions compared to the code in Figure 3. However, the soundness is preserved, since the assertion on line 12 couldn't be safely eliminated and is going to fail again: According to DSA, pointer `s1` is going to point to the field `a` of structure `S2`, and therefore its type is going to be `$S2#a` and not `$S1#x` as expected by the memory access.

The algorithm essentially compares compile-time pointer types used by Burstall's memory model with the sound over-approximation of the run-time types that DSA generates: if the two agree, we can safely omit the type check; if not, which could happen either because of actual type-unsafe casts or because of the imprecision of DSA, the type check stays. To sum up, using the extremely fast, cheap, and yet relatively precise Data Structure Analysis, we are eagerly getting rid of most of the type checks that are usually hard and expensive to prove later on.

In order for the remaining assertions to be discharged, either the user has to provide additional manual annotations that will essentially unify the types, which is the approach taken in some related work [12, 25], or such types can be unified automatically, which is our approach described in the next section.

```
1 const unique $S1#x:type;
2 const unique $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
9   s1 := s2;
10
11  Mem[$S2#a,s2] := Ptr(null,3);
12  assert(Type[s1] == $S1#x); // Fails!
13  Mem[$S1#x,s1] := Ptr(null,4);
14
15  assert(Mem[$S2#a,s2] == Ptr(null,3));
16 }
```

Fig. 5. Translation of Fig. 2 using the eager type check elimination algorithm. Compared to Fig. 3, the unneeded type checks have been eliminated, but the type-safety violation will still be caught.

3.3 Eager Type Unification

The type check elimination algorithm from the previous section doesn't remove the type check assertion for which the compile-time type of a pointer and the one computed by DSA don't agree. Proving those left-over assertions might still require the addition of manual annotations by a user. Instead, we describe a simple, completely automatic technique that will soundly remove the left-over assertions.

For each memory access for which the type check elimination algorithm couldn't agree on types, we unify the two types. Unification simply means that the type constants are not unique anymore, which is in BoogiePL achieved by removing the keyword `unique`. There is an obvious tradeoff between the type check elimination algorithm and the type unification algorithm: the first one might require additional running time and manual annotations from a user to discharge the left-over assertions; the second one is completely automatic, but with each unification, the memory model is closer to the monolithic one and the performance might suffer (in the worst case, all types are unified and we essentially have the monolithic model).

Figure 6 shows the translation using the eager type unification algorithm. Instead of the type-check assertion on line 12 in Figure 5, the types `$S1#x` and `$S2#a` are unified and are not unique constants any more (lines 1 and 2). Now, `Mem[$S2#a,s2]` and `Mem[$S1#x,s1]` possibly refer to the same location, which is sound, and therefore the assertion on line 14 will fail. Note that only the types `$S1#x` and `$S2#a` involved in the actual type-unsafe access got unified, while the type `$S2#b` not involved in type-unsafe operations didn't. Therefore, the overapproximation caused by unification is localized only to the places that actually need it in order to preserve soundness. In the limit, eager type unification degenerates into the monolithic memory model, but for code that is mostly type-safe, it should have most of the efficiency of Burstall's model and the soundness of the monolithic model.

```
1 const $S1#x:type;
2 const $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
9   s1 := s2;
10
11  Mem[$S2#a,s2] := Ptr(null,3);
12  Mem[$S1#x,s1] := Ptr(null,4);
13
14  assert(Mem[$S2#a,s2] == Ptr(null,3));
15 }
```

Fig. 6. Translation of Fig. 2 using the eager type unification algorithm. Instead of flagging the type-safety violation, this translation handles type unsafety by allowing `$S1#x` and `$S2#a` to be possibly the same type. Thus, the verifier will correctly catch the assertion violation on line 14.

3.4 Experimental Results

The results in Table 2 compare the running times for checking correct locking behavior while ensuring soundness using the three different approaches: guarding memory accesses with type assertions, eagerly eliminating type check assertions, and eagerly unifying types. The algorithm that inserts type checks for each memory access is a simple linear scan of the code and is extremely fast. Also, DSA scales to hundreds

Table 2. Total running times for checking correct locking behavior while ensuring soundness in Linux device drivers. The column "Every Access" gives the total running time of BOOGIE when checking type assertions on every access; "Eager Elimination" gives the total running time of BOOGIE when our eager elimination technique is used to soundly remove most of the required type checks; "Eager Unification" gives the total running time of BOOGIE when our eager unification technique is used to ensure soundness; "Speedup EA/EE" compares the running times of Every Access vs Eager Elimination; "Speedup EA/EU" compares the running times of Every Access vs Eager Unification. The * indicates that BOOGIE timed out on four and the memory blew up on one procedure from the applicom driver (time out is set to 1200s).

Driver	Assuring Soundness			Speedup	Speedup
	Every Access (s)	Eager Elimination (s)	Eager Unification (s)	EA/EE	EA/EU
ib700wd	448.7	14.2	14.1	31.6	31.8
w83877f_wdt	683.5	15.3	15.2	44.7	45.0
sc520_wdt	632.5	16.7	16.0	37.9	39.7
machzwd	761.4	18.2	17.8	41.8	42.8
wdt977	466.2	18.1	18.3	25.8	25.5
ds1286	823.5	20.7	25.5	39.8	32.4
efirtc	576.2	15.5	15.3	37.2	37.7
applicom	*7487.4	173.5	172.0	43.2	43.5

of thousands of lines of code in less than 4s [23]. Therefore, total running times are dominated by the verification done by BOOGIE, and those are the times we report.

As expected, blindly generating type check assertions for each memory access simply does not scale — verification times after using both eager techniques are roughly 30-40 times faster. Furthermore, both eager techniques give roughly the same verification times afterwards. The reason is, to our surprise, that none of the analyzed device drivers actually has type-unsafe structure casts. Therefore, both of the algorithms end up generating the same BoogiePL code. In the future, as we verify increasingly complex examples, we will be able to evaluate the trade-off between the two methods.

Bugs Found. While doing the experiments, we found a total of four bugs in the eight device drivers we checked from the Linux kernel. One bug is the rediscovered incorrect locking pattern in the `ds1286` driver that was also found earlier by the DDVERIFY checker. The other three bugs are previously unreported buffer-overflow bugs. We submitted the bugs to the Linux kernel development team, who confirmed all three bugs and issued patches to the standard Linux kernel.

A natural question is how the different memory models affected the detectability of these bugs. The answer is not straightforward:

- First, as mentioned above, these device drivers turned out to be type-safe, in the sense that Burstall's model would be as accurate as the monolithic model. Thus, one might argue that the more accurate models are unnecessary. However, the type-safety is not at all obvious — this is C code, with type casts, pointer arithmetic, etc. With Burstall's model, we assume type safety and might catch some bugs, but we don't know whether the code is truly type-safe; with the monolithic model, we don't assume type safety, but the verification complexity blows up, so we can't catch any bugs anyway. Our new models ensure type safety but also scale well.
- The other issue is that, of the four bugs we found, only the previously discovered one was a direct violation of the locking-unlocking properties we were checking. The other three bugs were buffer-overflow bugs that were caught because of the type-checking assertions. These bugs perhaps could have been caught by a variety of methods, using many different memory models.

The key point is that our new memory models can ensure, rather than assume, type-safety, yet are scalable enough to handle real code that is sufficiently complex to contain significant bugs that have eluded previous detection.

4 Conclusion and Future Work

In the first part of the paper, we presented our experience with two memory models for low-level code. We introduced the monolithic memory model, which can handle soundly many common low-level idioms. Then, we presented Burstall's memory model, which has typically been used in the verification of type-safe languages. We implemented both models as part of our verification tool SMACK. In the experiments, we checked correct locking behavior of a number of Linux device drivers, and showed that the performance using Burstall's model is much better than using the monolithic memory model, especially on more complex examples.

However, a straightforward translation of a program using Burstall's memory model cannot preserve soundness of type-unsafe operations found in low-level code. Therefore, in the second part of the paper, we describe three different techniques for ensuring soundness with Burstall's model: insertion of soundness checks before each memory access, our novel eager type check elimination algorithm based on a lightweight pointer analysis, and our novel eager type unification technique. We showed in the experiments that naively inserting checks is an unnecessary verification overhead, since most of the checks can be eagerly removed using our algorithms. During the verification effort, we found three previously unreported bugs.

In an upcoming paper [12], Condit et al. describe a novel memory model for low-level code that includes type information. Types can be checked using an SMT solver, and they also provide a decision procedure for checking type safety. Using these techniques, they type-checked a number of Windows device drivers. Their work is complementary to ours: we conservatively and eagerly remove as many type checks as possible, whereas they provide an efficient technique to prove type checks. Obvious future work is to combine the best of both approaches: quickly eliminating most type checks using our methods, and solving the remaining ones efficiently using theirs.

References

1. The DDVerify verification tool (2007) (Cited: August 15, 2008), `http://www.verify.ethz.ch/ddverify/`
2. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 203–213 (2001)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Workshop on Program Analysis For Software Tools and Engineering (PASTE), pp. 82–87 (2005)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Grégoire, B., Huisman, M., Lanet, J.-L. (eds.) CASSIS 2005. LNCS, vol. 3956, pp. 49–69. Springer, Heidelberg (2006)
6. 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: ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 196–207 (2003)
7. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, 23–50 (1972)
8. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: Intl. Conf. on Software Engineering (ICSE), pp. 385–395 (2003)
9. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)
10. Clarke, E., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
11. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design 25(2-3), 105–127 (2004)

12. Condit, J., Hackett, B., Lahiri, S., Qadeer, S.: Unifying type checking and property checking for low-level code. In: ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL (to appear, 2009)
13. Condit, J., Harren, M., Mcpeak, S., Necula, G.C., Weimer, W.: Cured in the real world. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 232–244 (2003)
14. Currie, D.W., Hu, A.J., Rajan, S., Fujita, M.: Automatic formal verification of DSP software. In: 37th Design Automation Conference, pp. 130–135. ACM/IEEE (2000)
15. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
16. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (2005)
17. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18, 453–457 (1975)
18. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
19. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), pp. 58–70 (2002)
21. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-Soft. In: Intl. Conf. on Computer Design (ICCD), pp. 297–308 (2005)
22. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Symp. on Code Generation and Optimization (CGO), pp. 75–88 (2004)
23. Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 278–289 (2007)
24. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 54–63 (2006)
25. Moy, Y.: Union and cast in deductive verification. In: C/C++ Verification Workshop (CCV), pp. 1–16 (2007)
26. Rakamarić, Z., Hu, A.J.: Automatic inference of frame axioms using static analysis. In: IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE), pp. 89–98 (2008)
27. Schulte, W., Xia, S., Smans, J., Piessens, F.: A glimpse of a verifying C compiler (extended abstract). In: C/C++ Verification Workshop (2007)
28. Siff, M., Chandra, S., Ball, T., Kunchithapadam, K., Reps, T.W.: Coping with type casts in C. In: European Software Engineering Conf./ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE), pp. 180–198 (1999)
29. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE), pp. 501–504 (2007)

Synthesizing Switching Logic Using Constraint Solving*

Ankur Taly[1], Sumit Gulwani[2], and Ashish Tiwari[3]

[1] Computer Science Dept., Stanford University
ataly@stanford.edu
[2] Microsoft Research, Redmond, WA 98052
sumitg@microsoft.com
[3] SRI International, Menlo Park, CA 94025
tiwari@csl.sri.com

Abstract. A new approach based on constraint solving techniques was recently proposed for verification of hybrid systems. This approach works by searching for inductive invariants of a given form. In this paper, we extend that work to automatic synthesis of safe hybrid systems. Starting with a multi-modal dynamical system and a safety property, we present a sound technique for synthesizing a switching logic for changing modes so as to preserve the safety property. By construction, the synthesized hybrid system is well-formed and is guaranteed safe. Our approach is based on synthesizing a controlled invariant that is sufficient to prove safety. The generation of the controlled invariant is cast as a constraint solving problem. When the system, the safety property, and the controlled invariant are all expressed only using polynomials, the generated constraint is an $\exists\forall$ formula in the theory of reals, which we solve using SMT solvers. The generated controlled invariant is then used to arrive at the maximally liberal switching logic.

1 Introduction

Formal verification is beginning to play an important role in the process of building reliable and certifiable complex engineered systems. A different approach to building correct systems is to automatically synthesize safe systems. The synthesis approach is attractive since it generates correct systems by design. However, computationally, the synthesis problem appears to be much harder than the verification problem and there are few general approaches for solving it.

Recently, Gulwani and Tiwari [7] introduced an approach for verification of (hybrid) systems that reduces the safety verification problem to satisfiability of $\exists\forall$ formulas over some theory (the theory of reals). Their method is based on finding an inductive invariant that proves the safety of the system. The "unbounded" search for invariants is "bounded" by fixing some templates for the

* Research supported in part by the National Science Foundation under grant CNS-0720721 and by NASA under Grant NNX08AB95A. Work done when the first author was visiting SRI International.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 305–319, 2009.

invariants. The existence ($\exists$) of an appropriate instance of the template that is also an inductive invariant ($\forall$) naturally maps to an $\exists\forall$ formula. If the $\exists\forall$ formula is valid (over the underlying theory), then it means that there exists an inductive invariant (of the form of the chosen template) that proves safety.

In theory, the constraint-based approach for verification described in Gulwani and Tiwari [7] can be generalized to solving the synthesis problem as well. Given an under-specified system, we can choose templates for the unknown parts of the system and the unknown inductive invariant. We can then obtain a $\exists\forall$ formula where all the unknowns are existentially quantified. The constraint solver then searches for instances of *all* these unknowns so that the resulting system is proved safe by the resulting invariant. In practice, however, this naive approach does not work well for synthesis because the constraint solver often chooses values that result in a degenerate system (such as, a zeno system, or a deadlocked system) where the safety property is vacuously true. Moreover, the above method does not take advantage of the correlations that exist between the various unknowns and uses a separate template for each unknown. Having too many templates contributes to the incompleteness and reduces the effectiveness of the approach.

In this paper, we define a specific instance of the synthesis problem, called the *switching logic synthesis* problem. We present a constraint-based approach, inspired by [7], to solve the switching logic synthesis problem. The novelty in our approach here is that we do not search for the switching conditions directly. Instead we use constraint solving to find an *inductive controlled invariant* set. Hence we only have to choose a single template – for the inductive control invariant – and none for the unknown switching conditions. In a final postprocessing step, we use the generated controlled invariant to synthesize the actual switching logic. This postprocessing step generates the weakest (most general) possible controller from the controlled invariant. Our approach is guaranteed to synthesize a non-blocking hybrid system that is also safe.

Inductive Controlled Invariant. An invariant for a system is any superset of the set of reachable states of that system. Safety properties can be proved by finding suitable invariants. However, invariance is difficult to check in general. A better alternative is to search for *inductive* invariants. Inductive invariants are attractive because inductiveness is a "local" property – for each state in the inductive set, we only need to check that the *immediate next* states reached from that state (rather than *all* reachable states) are also in the inductive set. Fortunately, the set of reachable states is always inductive and hence, the use of inductive invariants is a sound and complete method for safety verification.

In this paper, we consider systems that contain controllable choices, that is, the user/controller can make selections to achieve some safety goal. For such systems, the notion corresponding to invariant sets is called *controlled invariant.* A controlled reach set is the set of reachable states obtained for some choice of the controller. A controlled invariant is a superset of *some* controlled reach set. As before, the computationally interesting notion is that of an *inductive* controlled invariant. We can, therefore, synthesize safe controllers by generating the correct inductive controlled invariant. In this paper, we pursue this idea in

the context of hybrid systems, though the idea of inductive controlled invariant is applicable more generally.

Contribution and Outline of the Paper. In this paper, we present a formalization of the notion of inductive controlled invariants for multi-modal systems and describe a sound and complete approach for synthesizing switching logic from an inductive controlled invariant. (Section 3). Our synthesis technique relies on the deductive verification approach and does not use the usual game theoretic approach for controller synthesis, or the controlled reachability approach (See Section 7 for more discussion). We also describe several sufficient conditions for a set to be an inductive controlled invariant set. These conditions enable practical implementations for synthesizing controllers using template-based techniques (Section 4). Finally, we describe some heuristics to generate large controlled invariant sets, that lead to synthesis of the weak controllers (Section 5). We have performed preliminary experimental evaluation of our approach and presented some of the results as examples in the paper. We start by formally describing and motivating the problem in Section 2.

2 The Switching Logic Synthesis Problem

In this section, we describe the synthesis problem considered in this paper. We motivate our formal definitions with informal descriptions of the problem.

We are interested in controlling multi-modal continuous dynamical systems. A dynamical system is defined by its state-space, which is the set of all possible configurations/states of the system, and its dynamics, which defines how the system changes states (with time). Formally, a *continuous dynamical system* is a tuple $\langle X, f \rangle$ where X is a finite set of real-valued variables that define the state space $\mathbb{R}^X$ and $f : \mathbb{R}^X \mapsto \mathbb{R}^X$ is a vector field that specifies the continuous dynamics (as $\frac{d\boldsymbol{x}}{dt} = f(\boldsymbol{x})$). We assume that f is Lipschitz, which guarantees the existence and uniqueness of solutions to the ordinary differential equations.

Proposition 1 (Theorem 2.3.1, p80 [4]). *Consider a Lipschitz vector field f and the differential equation $\frac{dF(t)}{dt} = f(F(t))$, $F(t) = \boldsymbol{x_0}$. The solution of this differential equation, denoted by $F(\boldsymbol{x_0}, t)$, always exists and is unique. Moreover, $F(\boldsymbol{x_0}, t)$ depends continuously on the initial state $\boldsymbol{x_0}$.*

Often a single ordinary differential equation is insufficient to describe the system. Many systems have multiple modes and they have different dynamics in each mode. This happens, for example, when we introduce actuators inside physical devices that change the device's dynamics. In such cases, the dynamics of a system is described by a *collection* of differential equations. We call such system multi-modal dynamical systems. A multi-modal system has a finite number of different modes and in each mode, it behaves like a different continuous dynamical system. For instance, consider the water level in a tank with an inflow valve. Such a system has two dynamics – one when the valve is closed and one when it is open. Formally, we define a multi-modal continuous dynamical system (`MDS`) and its semantics as follows.

Definition 1 (Multi-modal Continuous Dynamical System). *A* multi-modal continuous dynamical system*, MDS, is a tuple $\langle X, f_1, f_2, \ldots, f_k, \mathit{Init}\rangle$, where $\langle X, f_i\rangle$ is a continuous dynamical system (representing the i-th mode) and $\mathit{Init} \subseteq \mathbb{R}^X$ is the set of initial states. Given an initial state $\boldsymbol{x_0} \in \mathit{Init}$, we say that a function $\boldsymbol{x}(t) : [0, \infty) \to R^X$ is a* trajectory *for MDS, if there is an increasing sequence $0 \leq t_1 < t_2 < \cdots$ (either finite or diverging to ∞) such that*

- *$\boldsymbol{x}(0) = \boldsymbol{x_0}$ and $\boldsymbol{x}(t)$ is continuous over $t \geq 0$, and*
- *for each interval (t_i, t_{i+1}), there is a mode $j \in I$ such that $\boldsymbol{x}(t)$ is smooth and $\frac{d\boldsymbol{x}(t)}{dt}(t') = f_j(\boldsymbol{x}(t'))$ for all t in the range $t_i < t < t_{i+1}$. When $i = 0$, then we require $j = 1$; that is, mode 1 is the initial mode.*

Following Definition 1, a multi-modal system can nondeterministically switch between its modes. However, switching between the different modes in a multi-modal dynamical system is often controllable. The goal of controlling a system is to achieve safe operation with some desired performance. For instance, in the water tank example, the transition between the two modes can be controlled by opening and closing the valve. The controller may be required to guarantee that the water level in the tank remains between two thresholds. There are several controllers that can achieve this property. A controller that opens the valve just when the water level reaches the lower threshold and closes it soon thereafter, will keep the level closer to the lower threshold, but it is very restrictive as it prevents the system from reaching several possible safe states. We are interested in designing controllers that guarantee safety, but that also do not unnecessarily restrict the system from reaching safe states.

A controller for a multi-modal system is specified as a *switching logic*.

Definition 2 (Switching Logic). *A* switching logic *SwL for a multi-modal dynamical system $\mathit{MDS} := \langle X, (f_i)_{i\in I}, \mathit{Init}\rangle$ is a tuple $\langle (g_{ij})_{i\neq j; i,j\in I}, (\mathit{Inv}_i)_{i\in I}\rangle$, containing guards $g_{ij} \subseteq \mathbb{R}^X$ and state (location) invariants $\mathit{Inv}_i \subseteq \mathbb{R}^X$.*

Informally, the guard g_{ij} specifies the condition under which the system *could* switch from mode i to mode j and the state invariant $\mathtt{Inv}_i$ specifies the condition which *must* be respected while in mode i.

A multi-modal system `MDS` can be combined with a switching logic `SwL` to create a hybrid system $\mathtt{HS} := \mathtt{HS}(\mathtt{MDS}, \mathtt{SwL})$ in the following natural way: the hybrid system `HS` has $\|I\|$ modes with dynamics given by $\frac{dX}{dt} = f_i$ in mode i, and with g_{ij} being the guard on the discrete transition from mode i to mode j and $\mathtt{Inv}_i$ being the state invariant in mode i. The initial states are $\{1\} \times \mathtt{Init}$. The discrete transitions in `HS` have identity reset maps, that is, the continuous variables do not change values during discrete jumps. The semantics of hybrid systems that define the set of reachable states of hybrid systems are standard [1].

Though semantically well-defined, some hybrid systems have undesirable behaviors. For example, it can happen that a hybrid system, in mode i, reaches a point $\boldsymbol{x}$ on the boundary of $\mathtt{Inv}_i$, but there is no valid trajectory from $\boldsymbol{x}$; that is, there is no discrete transition enabled at $\boldsymbol{x}$, and following mode i dynamics takes the system out of $\mathtt{Inv}_i$. The non-blocking requirement disallows such cases. We are interested in synthesizing non-blocking hybrid systems.

Definition 3. *A hybrid system* HS *is said to be* non-blocking *if for every mode i, and for every point $\boldsymbol{x}$ on the boundary of the state invariant for mode i, there exists a mode j (may be same as i) and $\epsilon > 0$ such that (i) $\boldsymbol{x} \in g_{ij}$ whenever $i \neq j$, and (ii) the dynamics of mode j keeps the system within the state invariant of mode j for at least ϵ time.*

A hybrid system HS is safe with respect to a safety property $\texttt{Safe} \subseteq \mathbb{R}^X$ if the set of its reachable states is contained in Safe. Formally, we define the logic synthesis problem as follows:

Definition 4 (Switching Logic Synthesis Problem). *Given a multi-modal dynamical system $\mathit{MDS} := \langle X, f_1, f_2, \ldots, f_k, \mathit{Init}\rangle$ and a safety property $\mathit{Safe} \subseteq \mathbb{R}^X$, the* switching logic synthesis problem *seeks to synthesize a switching logic SwL such that the hybrid system HS(MDS, SwL) is safe with respect to Safe.*

3 The Synthesis Procedure

In this section we present a high-level procedure for solving the switching logic synthesis problem described in Definition 4. We fix our notation and denote the given multi-modal dynamical system by MDS, its initial set of states by Init and the given safety property by Safe.

We first define the notion of a controlled invariant set.

Definition 5 (Controlled Invariant). *A set CInv is said to be a* controlled invariant *for a $\mathit{MDS} := \langle X, (f_i)_{i\in I}, \mathit{Init}\rangle$ if for all $\boldsymbol{x}_0 \in \mathit{Init}$, there exists a trajectory (Definition 1) $\boldsymbol{x}(t)$ such that $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and for all $t \geq 0$, $\boldsymbol{x}(t) \in \mathit{CInv}$.*

Note that an invariant requires that *every* trajectory (starting from an initial state) remains inside the invariant. In contrast, a controlled invariant only requires *some* trajectory remains inside the controlled invariant.

Example 1. Let $\dot{x}$ denote $\frac{dx}{dt}$. Consider a multi-modal system with two modes. In mode 1, $\dot{x} = 1, \dot{y} = 0$, while in mode 2, $\dot{x} = 0, \dot{y} = 1$. If $x = 0, y = 0$ is the only initial state, then $x \geq 0 \wedge y \geq 0$ is an invariant, whereas $x \geq 0 \wedge y = 0$ is a controlled invariant that is not an invariant. The set $x + y \leq 0$ is not a controlled invariant.

Definition 5 does not suggest any easy way to compute nontrivial controlled invariants. Hence, we define the notion of *inductive* controlled invariants. Since the dynamics are continuous here, we first need to define a few notions. Recall that the vector fields f_i's are Lipshitz and hence, by Proposition 1, we have a unique trajectory $F_i(\boldsymbol{x_0}, t)$ in mode i. By $F_i(\boldsymbol{x_0}, (0, \epsilon))$ we denote the set of all points reached in the time interval $(0, \epsilon)$; that is, $F_i(\boldsymbol{x_0}, (0, \epsilon)) := \{\boldsymbol{x} \mid \boldsymbol{x} = F_i(\boldsymbol{x_0}, t), 0 < t < \epsilon\}$. For a set $S \subseteq \mathbb{R}^n$, let ∂S denotes the boundary of S in the topological sense. We are now ready to define inductive controlled invariants.

Definition 6 (Inductive Controlled Invariant). *A closed set CInv is an* inductive controlled invariant *for $\mathit{MDS} := \langle X, (f_i)_{i\in I}, \mathit{Init}\rangle$ if*

(A1) $\mathit{Init} \subseteq \mathit{CInv}$ *and*

(A2) $\forall \boldsymbol{x} \in \partial \mathit{CInv} : \exists i \in I : \exists \epsilon > 0 : F_i(\boldsymbol{x}, (0, \epsilon)) \subseteq \mathit{CInv}$

```
SynthSwitchLogic(MDS, Safe) :
1. Find a closed set CInv that satisfies Conditions (A1) and (A2)
   from Definition 6 and Condition (A3) below
     (A3)  CInv ⊆ Safe
   If no such set is found, return failure
2. Let bdry_i := {x ∈ ∂CInv | ∃ε > 0 : F_i(x, (0, ε)) ⊆ CInv} for all i ∈ I
3. Let Inv_i := CInv for all i ∈ I
4. Let g_ij := bdry_j ∪ Interior(CInv) for all i ≠ j; i, j ∈ I,
Return SwL := ⟨(g_ij)_{i≠j;i,j∈I}, (Inv_i)_{i∈I}⟩
```

Fig. 1. Procedure for synthesizing switching logic presented at a semantic level

Intuitively, Condition (A2) in Definition 6 says that for every point on the boundary of CInv, there is a vector field f_i that points inwards and brings the system (instantaneously) inside the set CInv, see also [3]. Just as inductive invariants are also invariants, inductive controlled invariants are also controlled invariants.

Proposition 2. *If a closed set CInv is an inductive controlled invariant for MDS, then it is also a controlled invariant for MDS.*

The complete procedure, at a semantic level, for solving the switching logic synthesis problem is presented in Figure 1. The key idea behind the synthesis procedure is to find an inductive controlled invariant set CInv and then design the guarded transitions so that the resulting hybrid system always remains in CInv. Conditions (A1), (A2), and (A3) imply that CInv is an inductive controlled invariant that proves safety. It follows from the definition of CInv that its boundary ∂CInv can be written as a union

$$\partial\texttt{CInv} = \bigcup_{i \in I} \texttt{bdry}_i \tag{1}$$

such that $\forall \boldsymbol{x} \in \texttt{bdry}_i$, it is the case that $\exists \epsilon > 0 : F_i(\boldsymbol{x}, (0, \epsilon)) \subseteq \texttt{CInv}$. This fact is used to define the sets $\texttt{bdry}_i$ in Line 2. In Line 4, we use the sets $\texttt{bdry}_i$ and CInv to define the guards for the various discrete transitions.

We next state and prove some properties of the procedure SynthSwitchLogic in Figure 1. We show that the synthesized hybrid system is always non-blocking and safe (soundness). Furthermore, if there is a safe hybrid system, then under some fairly general conditions, the procedure SynthSwitchLogic will return a switching logic SwL and synthesize a safe system HS(MDS, SwL) (completeness).

Theorem 1 (Soundness). *For every switching logic SwL returned by procedure SynthSwitchLogic, the hybrid system HS(MDS, SwL) is non-blocking and safe.*

We prove completeness under a technical assumption. We say a hybrid system HS has the min-dwell-time property if there exists a fixed time duration t_a such that for all reachable states $\boldsymbol{x}$, if the hybrid system permits a mode switch from i to j at $\boldsymbol{x}$, then there must exist a mode k such that the hybrid system permits a mode switch from i to k at $\boldsymbol{x}$ and the system can stay in mode k for at least t_a

units of time starting at $\boldsymbol{x}$. The min-dwell-time property implies that successive mode switchings can be forced to be t_a units apart.

Theorem 2 (Completeness). *For any switching logic* `SwL`*, if a hybrid system* $\mathtt{HS} = \mathtt{HS}(\mathtt{MDS}, \mathtt{SwL})$ *is safe,* `HS` *satisfies the min-dwell-time property and* `Safe` *is a closed set, then procedure* `SynthSwitchLogic` *will return a switching logic.*

Although the above procedure is sound and complete, it is not computationally feasible as there is no easy way to check for Condition `(A2)`. In the next section we will replace Condition `(A2)` by something stronger that can be easily computed. This causes loss of completeness, but it preserve soundness.

4 Implementing the Procedure

The procedure for solving the switching logic synthesis problem was described at a semantic level in the previous section. In this section, we show how that procedure can be concretely implemented.

Recall that a set `CInv` is an inductive controlled invariant if it satisfies Conditions `(A1)` and `(A2)`. Condition `(A2)` is not easy to check as F_i's are solutions of differential equations. We solve this problem by replacing this condition by a stronger condition, `(B2)`, which, as we show later, can be tested *without* explicitly computing F_i. Let $\mathtt{Interior}(\mathtt{CInv}) := \mathtt{CInv} - \partial\mathtt{CInv}$. We ensure that `CInv` is an inductive controlled invariant (that proves safety) by checking:

`(B1)` $\mathtt{Init} \subseteq \mathtt{CInv}$

`(B2)` $\forall \boldsymbol{x} \in \partial\mathtt{CInv} : \exists i \in I : \exists \epsilon > 0 : F_i(\boldsymbol{x}, (0, \epsilon)) \subseteq \mathtt{Interior}(\mathtt{CInv})$

`(B3)` $\mathtt{CInv} \subseteq \mathtt{Safe}$

We will now present a condition that is equivalent to `(B2)` and that can be easily computed. We first need to fix a representation for `CInv`.

We use semi-algebraic sets as candidates for $\mathtt{CInv} \subseteq \mathbb{R}^X$. Since `CInv` is unknown, we use the idea of templates. A template is a formula (in the theory of reals) with free variables $X \cup U$. Here U are the (real-valued) unknown coefficients that need to be instantiated to yield the desired `CInv`. We use boolean combinations of polynomial equalities and inequalities (semi-algebraic sets) as the formulas. Once a template is fixed, we can write Conditions `(B1)`, `(B2)` and `(B3)` as an $\exists\forall$ formula over the theory of reals [7]. Concretely, let $p(U, X)$ be a polynomial and $p(U, X) \geq 0$ be the chosen template for searching for `CInv`. We restrict ourselves to the case of a single inequality $p(U, X) \geq 0$ for simplicity of presentation. For example, $u_1x_1 + u_2x_2 \geq u_3$ is a linear template over 2 variables $X = \{x_1, x_2\}$ and 3 unknown coefficients $U = \{u_1, u_2, u_3\}$. The following formula states that there is a choice of values for U such that the resulting set, $p(U, X) \geq 0$, is a controlled invariant sufficient to prove safety.

$$\exists U : \forall X : \ (X \in \mathtt{Init} \Rightarrow p(U, X) \geq 0) \ \wedge \ (p(U, X) \geq 0 \Rightarrow X \in \mathtt{Safe}) \ \wedge \\ (p(U, X) = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p(U, X) > 0) \tag{2}$$

Here $\mathcal{L}_{f_i}p$ denotes the derivative of p with respect to time t and is called the Lie derivative of p with respect to the vector field f_i. It can be symbolically computed using the chain rule as,

$$\mathcal{L}_{f_i}p := \sum_{x \in X} \frac{\partial p}{\partial x}\frac{dx}{dt}.$$

Note that we have used a test on the Lie derivatives to encode Condition `(B2)`. This test is equivalent to `(B2)` and allows us to verify it without requiring F_i.

Remark 1. It is tempting to think that replacing $>$ by $\geq$ in Formula (2) will make the formula equivalent to checking Condition `(A2)`, but this is not true.

If each of the vector fields, f_i, is specified using polynomials (i.e., in each mode, $\frac{dX}{dt}$ is a vector of polynomials), then $\mathcal{L}_{f_i}p$ is simply a polynomial. If `Init` and `Safe` are semi-algebraic sets, then the membership tests ($X \in$ `Init` and $X \in$ `Safe`) can also be written as formulas using only polynomials. Thus Formula (2) is a $\exists\forall$ formula consisting only of polynomial expressions.

Corollary 1. *If Formula (2) is valid in the theory of reals, then there is a controlled invariant* `CInv` *that proves safety.*

Corollary 1 immediately gives us a sound procedure that reduces the switching logic synthesis problem to solving of an $\exists\forall$ constraint in the theory of reals. We illustrate the procedure on the following example.

Example 2. Consider a train gate controller with two modes: In the *about to lower* mode (1), distance x of the train from the gate decreases according to $\dot{x} = -50$ and the gate angle g does not change. In the *gate lowering* mode (2), we have $\dot{x} = -50$ and $\dot{g} = -10$. The initial state is $g = 90 \wedge x = 1000$. We wish to synthesize the switching logic so that the system always stays in the safe region $x > 0 \vee g \leq 0$. We assume a template of the form $x + a_1 g \geq a_2$ for the controlled invariant. Writing out Formula (2), we get:

$$\begin{array}{ll}
\exists a_1, a_2 : \forall x, g : & \\
\quad (x = 1000 \wedge g = 90 \;\Rightarrow\; x + a_1 g \geq a_2) \wedge & \text{(Condition (B1))} \\
\quad (x + a_1 g \geq a_2 \;\Rightarrow\; x > 0 \vee g \leq 0) \wedge & \text{(Condition (B3))} \\
\quad (x + a_1 g = a_2 \;\Rightarrow\; -50 + 0 > 0 \vee -50 - 10a_1 > 0) & \text{(Condition (B2))}
\end{array}$$

Our solver returns $a_1 = -10, a_2 = 50$; that is, we get $x - 10g \geq 50$ as the controlled invariant. The resulting hybrid system has $x - 10g \geq 50$ as the state invariant for each mode. The guards for transitions are $g_{12} = x - 10g \geq 50$ (as dynamics for mode 2 points inwards everywhere on the boundary) and $g_{21} = x - 10g > 50$ (dynamics for mode 1 never points inwards on the boundary, so no boundary point gets assigned to g_{21}). So, if the system starts in mode 1, it can continue in 1 until $x - 10g = 50$ is true, whence the system will have to shift to mode 2. The resulting hybrid system is safe and non-blocking. ■

4.1 A Variant Procedure

In the previous section, we approximated the semantic condition (A2) by the constraint $p = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p > 0$. As Corollary 1 shows, this is a sound approximation. However, the requirement that a vector field points *strictly* inwards, which is captured by $\mathcal{L}_{f_i} p > 0$, is too strong and leads to incompleteness, which leads to failure in finding suitable controlled invariant sets in practice. In this section, we weaken Formula (2) so that it can be used to handle more examples.

We weaken Condition (B2) and use the following weaker version of Formula (2) to test if $p(X, U) \geq 0$ is an inductive controlled invariant:

$$\exists U \forall X : \ (X \in \texttt{Init} \Rightarrow p(U, X) \geq 0) \ \wedge \ (p(U, X) \geq 0 \Rightarrow X \in \texttt{Safe}) \ \wedge \\ p(U, X) = 0 \Rightarrow \bigvee_{i \in I} (\mathcal{L}_{f_i} p > 0 \vee (\mathcal{L}_{f_i} p = 0 \wedge \bigwedge_{j \neq i} \mathcal{L}_{f_j} p < 0)) \qquad (3)$$

Formula (3) says that at the boundary ($p = 0$) of the controlled invariant ($p \geq 0$), either some vector field, say f_i, points strictly inwards ($\mathcal{L}_{f_i} p > 0$), or exactly one vector field is tangential ($\mathcal{L}_{f_i} p = 0$) and all others point strictly outside ($\mathcal{L}_{f_j} p < 0$). The counterintuitive condition – vector fields pointing strictly outwards – helps in proving that the tangential vector field will keep the system inside the controlled invariant.

We can now replace the constraint in Step (1) of the procedure in Figure 2 by Formula (3) and get a new and more powerful procedure for solving the switching logic synthesis problem. We can again prove soundness of the technique.

Corollary 2. *If Formula (3) is valid in the theory of reals, then there is a controlled invariant* `CInv` *that proves safety provided* $\|I\| > 1$.

Remark 2. The procedure could be unsound when $\|I\| = 1$. This unsoundness is related to the comment in Remark 1.

We illustrate the advantage of weakening the constraint for the inductive test by using the following example.

Example 3. Consider a system with continuous variable x and y and two modes. In mode 1, $\dot{x} = 0, \ \dot{y} = -1$ and in mode 2, $\dot{y} = 0, \ \dot{x} = -1$. The initial state

`SynthSwitchLogicImpl(MDS,Init,Safe)`
`0. Choose template for controlled invariant, say` $p(U, X) \geq 0$
`1. Generate` $\exists\forall$ `constraint for template to be a controlled invariant`
$\exists U : \forall X : (X \in \texttt{Init} \Rightarrow p \geq 0) \wedge (p \geq 0 \Rightarrow X \in \texttt{Safe}) \wedge (p = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p > 0)$
`1. Solve the` $\exists\forall$ `constraint and get values` $\boldsymbol{u}$ `for` U
`2. Let` $\texttt{bdry}_i := (p(\boldsymbol{u}, X) = 0 \ \wedge \ \mathcal{L}_{f_i} p > 0)$ `for all` $i \in I$
`3. Let` $\texttt{Inv}_i := (p(\boldsymbol{u}, X) \geq 0)$ `for all` $i \in I$
`4. Let` $g_{ij} := \texttt{bdry}_j \ \vee \ (p(\boldsymbol{u}, X) > 0)$ `for all` $i \neq j; i, j \in I$,
`Return SwL` $:= \langle (g_{ij})_{i \neq j; i, j \in I}, (\texttt{Inv}_i)_{i \in I} \rangle$

Fig. 2. A sound procedure for solving the switching logic synthesis problem

is $x = 10, y = 10$ and the desired safety property is $y \geq 0$. We start with the template $a_1x + a_2y \geq a_3$. Formula (3) then becomes:

$$
\begin{array}{l}
\exists a_1, a_2, a_3 : \forall x, g : \\
\quad (x = 10 \wedge y = 10 \Rightarrow a_1x + a_2y \geq a_3) \wedge \qquad \text{(Condition } \texttt{(A1)}\text{)} \\
\quad (a_1x + a_2y \geq a_3 \;\Rightarrow y \geq 0) \wedge \qquad \text{(Condition } \texttt{(A3)}\text{)} \\
\quad (a_1x + a_2y = a_3 \;\Rightarrow -a_1 > 0 \vee (-a_1 = 0 \wedge -a_2 < 0) \vee \\
\qquad\qquad\qquad -a_2 > 0 \vee (-a_2 = 0 \wedge -a_1 < 0)) \quad \text{(Condition } \texttt{(A2)}\text{)}
\end{array}
$$

We get a solution $a_1 = 0, a_2 = 1, a_3 = 1$. So the invariant obtained is $y \geq 1$. Note that on the boundary of the controlled invariant, the dynamics in mode 2 moves along the boundary and that of mode 1 points outwards. The previous method fails to find a controlled invariant for this example. ■

Example 4. Consider the train gate controller model from Example 2. Observe that the controller synthesized is very conservative and forces the system to switch from mode 1 to 2 in $t \leq 1$ units. Applying the variant procedure on this example, we get the following $\exists\forall$ formula:

$$
\begin{array}{l}
\exists a_1, a_2 : \forall x, g : \\
\quad (x = 1000 \wedge g = 90 \Rightarrow x + a_1g \geq a_2) \wedge \qquad \texttt{(A1)} \\
\quad (x + a_1g \geq a_2 \quad\;\; \Rightarrow x > 0 \vee g \leq 0) \wedge \qquad \texttt{(A3)} \\
\quad (x + a_1g = a_2 \quad\;\; \Rightarrow -50 > 0 \vee (-50 = 0 \wedge -50 - 10a_1 < 0) \vee \\
\qquad\qquad\qquad -50 - 10a_1 > 0 \vee (-50 - 10a_1 = 0 \wedge -50 < 0)) \quad \texttt{(A2)}
\end{array}
$$

This time the solver returned $a_1 = -5,\ a2 = 50$ as the solution, which gives $x - 5g \geq 50$ as the controlled invariant. So the resulting hybrid system has $x - 5g \geq 50$ as the state invariant for each mode and the guards $g_{12} = x - 5g \geq 50$ and $g_{21} = x - 5g > 50$ are computed. In this case, the switch from mode 1 to mode 2 could be delayed by as much as 10 units. ■

5 Synthesizing a Good Controller

In the previous section, two sound approaches were presented for solving the switching logic synthesis problem. Neither method gives any guarantee on the quality of the generated controller. A controller that minimally restricts the dynamics – and consequently results in a system with a maximal reach set – is preferable since it provides more opportunities for being refined later for other requirements. In this section, we present heuristics that improve the quality of solution generated by the two approaches presented in Section 4.

The size of the generated controlled invariant is a good measure of the quality of the solution. We desire to synthesize the largest possible inductive controlled invariant `CInv` because this would allow the maximal possible behaviors. It is not immediately clear how this can be achieved in our approach. Intuitively, the problem of finding the largest inductive controlled invariant is naturally seen as an *optimization* problem, whereas in our approach of using constraints, we are casting the problem as a *satisfiability* problem that asks for some solution and not the "best" solution. We now present three different ways to address the above problem.

5.1 Binary Search

The first solution for finding good controllers is based on iteratively searching for larger controlled invariants. In the first iteration, we use one of the methods from Section 4 to compute `CInv`. In each subsequent iteration, we add an additional constraint that forces search for a larger set `CInv`. For example, if we use the template $p(U, X) \geq 0$, and the first iteration returns the controlled invariant $p(\boldsymbol{u}, X) \geq 0$, then in the next iteration we use the template $p'(v, X) := p(\boldsymbol{u}, X) \geq v$ (containing only one parameter v) and add an additional constraint $v \leq -1$. If the second iteration is successful, then the controlled invariant generated in the second iteration will necessarily contain the controlled invariant generated in the first iteration. In the case when we know a lower bound on v, say $lb < 0$, then we can search for the optimal v by using a binary search in the interval $[lb, 0]$. This approach can be used to find the largest controlled invariant in the set $\{p(\boldsymbol{u}, X) \geq v \mid v \in [lb, 0],\ v \text{ an integer}\}$ in $O(\log \|lb\|)$ iterations.

5.2 Encoding Optimality Constraints Directly

We now present a different technique for capturing the optimality requirement. It is based on adding more constraints to the $\exists\forall$ formula. Intuitively, the new constraints say that at least one of the implications in the $\exists\forall$ formula is tight.

A reasonable heuristic for identifying if `CInv` is maximally large is to test if the boundary of `CInv` touches the boundary of the unsafe set $\overline{\texttt{Safe}}$. Hence, we introduce the following additional constraint in the original $\exists\forall$ formula:

$$\partial\texttt{CInv} \cap \partial\texttt{Cl}(\overline{\texttt{Safe}}) \neq \emptyset$$

This constraint can be written as an $\exists$ formula. Since we assume the sets `CInv` and `Safe` are given using polynomial inequalities, the boundaries of these sets can be expressed using polynomial equations and inequalities. The above constraint corresponds to tightening Condition (A2).

Example 5. Consider the train gate controller from example 4. The controlled invariant obtained by using the variant procedure on this example is $x - 10g \geq 50$. Observe that this is not the largest controlled invariant possible because when $x = 0$, this invariant implies $g \leq -5$, whereas safety just requires $g \leq 0$. If we add an additional constraint for tightening condition $A2$, which in this case is $\exists x_1, g_1 : x_1 + a_1 g = a_2 \wedge x = 0 \Rightarrow g = 0$, to the $\exists\forall$ formula, we get $x - 10g \geq 0$ as the controlled invariant. This is the largest controlled invariant for the template $x - 10g \geq v$. ■

Tightening Condition (A3). Before we describe the constraint for encoding tightness of Condition (A3), we need a few details on the procedure we use to solve the $\exists\forall$ formulas from [7]. The $\exists\forall$ formulas are solved in two steps. In the first step, the $\forall$ quantifier is eliminated and replaced by new $\exists$ quantifiers. The result of the first step is a purely existentially quantified formula which is solved using SMT solvers in the second step. The first step is achieved using a variant of Farkas Lemma – which is a technique for replacing $\forall$ by $\exists$ quantification.

Lemma 1. *It is the case that Formula (4) is implied by Formula (5).*

$$\exists U : \forall X : \ ((\bigwedge_j p_j = 0) \wedge (\bigwedge_k q_k > 0) \Rightarrow p \geq 0) \tag{4}$$

$$\exists U, \nu_j, \lambda_k, \lambda, \mu : \ \lambda_k \geq 0 \ \wedge \ \lambda \geq 0 \ \wedge \ \mu > 0 \ \wedge \ (\forall X : (\sum_p \nu_j p_j + \sum_k \lambda_k q_k + \lambda - \mu p = 0)) \tag{5}$$

Lemma 1 can be used to eliminate the internal $\forall X$ quantifier by noting that the polynomial in Formula (5) is zero for all X, if and only if, all coefficients of all power products of X in that polynomial are identically 0. We note that the term λ in Formula (5) is a "slack" term. If Formula (5) is satisfied when $\lambda = 0$, then we say that the implication of Formula (4) is *tight*.

Now consider Condition (A3) which encodes the boundary condition. In Section 4, this condition was approximately captured in Formula (2) and Formula (3). Using elementary logical manipulations, we can rewrite these formulas in the form

$$\exists U : \bigwedge_i (\forall X : (\bigwedge_j p_{ij} = 0) \wedge (\bigwedge_k q_{ik} > 0) \Rightarrow p_i \geq 0). \tag{6}$$

Apply Lemma 1 to each outer conjunct and let λ_i be the slack term for the i-th conjunct.

Now we are ready to state the constraint that enforces tightness on Condition (A3). This new constraint is not added to the $\exists\forall$ formula. It is added to the existential formula generated after the $\forall$ quantifiers have been eliminated using Lemma 1. The constraint we add is the following:

$$\phi_{opt} := \bigvee_i (\lambda_i = 0) \tag{7}$$

If the existential formula, with ϕ_{opt} added, is satisfiable and we get a controlled invariant $p(\boldsymbol{u}, X) \geq 0$, then we can show the obtained controlled invariant is the "best possible" among the set $\{p(\boldsymbol{u}, X) \geq \alpha \mid \alpha \in \mathbb{R}\}$.

Theorem 3 (Correctness). *Let $\boldsymbol{u}$ be a set of values for variables U that satisfy the existential formula $\phi_\exists \wedge \phi_{opt}$, where $\phi_\exists$ is the existential formula generated from Formula (2) (or Formula (3)) using Lemma 1. Then, there is no controlled invariant $p(\boldsymbol{u}, X) \geq \alpha$ for any $\alpha < 0$ that also satisfies the existential formula generated from Formula (2) (or Formula (3)) using $p(\boldsymbol{u}, X) \geq \alpha$ as a template.*

6 Extensions and Future Work

In our presentation so far, we have restricted all discussion, for simplicity, to simple templates of the form $p(U, X) \geq 0$. However, the two procedures described in

Section 4 can be generalized to the case when the template is a boolean combination of nonstrict polynomial inequalities. When the template is a conjunction, say $p_1 \geq 0 \wedge p_2 \geq 0$, then Formula (2) generalizes to

$$\begin{aligned}
&\exists U \forall X : (X \in \texttt{Init} \Rightarrow p_1 \geq 0 \wedge p_2 \geq 0) \ \wedge \ (p_1 \geq 0 \wedge p_2 \geq 0 \Rightarrow X \in \texttt{Safe}) \ \wedge \\
&\quad (p_1 = 0 \wedge p_2 > 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p_1 > 0) \ \wedge \ (p_1 > 0 \wedge p_2 = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p_2 > 0) \ \wedge \\
&\quad (p_1 = 0 \wedge p_2 = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p_1 > 0 \wedge \mathcal{L}_{f_i} p_2 > 0)
\end{aligned}$$

When the template is a disjunction, say $p_1 \geq 0 \vee p_2 \geq 0$, then Formula (2) generalizes to

$$\begin{aligned}
&\exists U \forall X : (X \in \texttt{Init} \Rightarrow p_1 \geq 0 \vee p_2 \geq 0) \ \wedge \ (p_1 \geq 0 \vee p_2 \geq 0 \Rightarrow X \in \texttt{Safe}) \ \wedge \\
&\quad (p_1 = 0 \wedge p_2 < 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p_1 > 0) \ \wedge \ (p_1 < 0 \wedge p_2 = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p_2 > 0) \ \wedge \\
&\quad (p_1 = 0 \wedge p_2 = 0 \Rightarrow \bigvee_{i \in I} \mathcal{L}_{f_i} p_1 > 0 \vee \mathcal{L}_{f_i} p_2 > 0)
\end{aligned}$$

We can similarly generalize Formula (3) for the case when the template is a disjunction or conjunction of polynomial inequalities. The following example illustrates this case.

Example 6. Consider a thermostat controller with two continuous variables temperature (t) and power (p) and two modes *on* and *off*. In the *on* mode, the dynamics is $\dot{p} = +1 \wedge \dot{t} = p - 10$ and in the *off* mode, it is $\dot{p} = -1 \wedge \dot{t} = p - 10$. The initial state is $p = 10, t = 75$ and the mode is *on*. The desired safety property is $70 \leq t \leq 80$. We start with the following conjunctive template for the controlled invariant: $a_1 p^2 + a_2 p + a_3 t + a_4 \geq 0 \wedge b_1 p^2 + b_2 p + b_3 t + b_4 \leq 0$. Using the generalization of Formula (3) to conjunctive templates, we get a $\exists\forall$ formula. Solving this formula, we get $a_1 = -1, a_2 = 20, a_3 = 2, a_4 = -172, b_1 = 1, b_2 = -20, b_3 = 100, b_4 = 23$ as one possible solution. This gives the invariant $\frac{-(p-10)^2}{2} + t \geq 72 \wedge \frac{(p-10)^2}{2} + t \leq 77$. The switching conditions can be obtained from the controlled invariant by using the procedure `SynthSwitchLogic`. It is easy to see that this is a safe controller. However it is not the most liberal controller. If we add an additional constraint to tighten the Condition (A2) (make the controlled invariant touch the boundary of the unsafe set), then we obtain $\frac{-(p-10)^2}{2} + t \geq 70 \wedge \frac{(p-10)^2}{2} + t \leq 80$ as the controlled invariant. In fact, this is the most liberal controller for this system. ■

The methods discussed in Section 5 can also be extended to the case of conjunctive and disjunctive templates, but we do not discuss the details here.

Our basic approach can be adapted to handle natural variants of the switching logic synthesis problem. First, note that we have assumed that each mode of the multi-modal system has the complete state space as its given state invariant. If the given modes have nontrivial state invariants, we can use them in our

constraints and the synthesized controller can potentially refine them. Second, our synthesized controller could have zeno behaviors. It appears that making the constraints stronger (as done in Section 4) already reduces the possibility of synthesizing zeno hybrid systems. This aspect needs further investigation.

We have a preliminary implementation of the approaches described in Section 4, along with the optimality variants of Section 5. The input is a multi-modal system, a safety property, and a template – all specified using only polynomials – and the output is a switching logic, if it exists. This implementation was used to solve the examples in the paper and their variants. We currently use the technique from [7] for solving the $\exists\forall$ constraints. Future work involves improving this technique using a symbolic nonlinear solver. This will enable applicability to larger and more complex examples. Our constraint-based technique relies heavily on the choice of the template. We currently start with linear or quadratic templates that have 1 to 4 conjuncts or disjuncts. It will be interesting to find classes of systems for which a given class of templates is complete.

7 Related Work

Constraint-based techniques have been used for safety verification of hybrid systems [7, 11, 12], wherein $\exists\forall$ constraints are generated from the user-provided invariant templates. The various approaches differ in the form of the invariants considered, the technique used to generate the $\exists\forall$ formula, and the approach for solving it. In this paper, we present a constraint-based technique for the synthesis problem that also involves generating and solving a $\exists\forall$ formula from template controlled invariants. The novelty of our work lies in the formalization of inductive controlled invariant approach for solving synthesis problem and showing that it can be reduced to solving $\exists\forall$ constraints.

There is a lot of work on synthesis of controllers for hybrid systems, which can be broadly classified into two categories. The first category finds controllers that meet some liveness specifications, such as synthesizing a trajectory to drive a hybrid system from an initial state to a desired final state [8, 9]. The second category finds controllers that meet some safety specification. Our work falls in this category. For a detailed discussion on the related work in this category, we refer the reader to Asarin et al. [2]. There are two main approaches for synthesis: direct approaches that compute the controlled reachable states in the style of solving a game [2, 13], and abstraction-based approaches that do the same, but on an abstraction or approximation of the system [6, 10]. Some of these approaches are limited in the kinds of continuous dynamics they can handle. They all require some form of iterative fixpoint computation. Our work here, based on synthesizing inductive controlled invariants, is an entirely different approach for controller synthesis that does not require any fixpoint computation.

There is a large body of work in the area of program synthesis. These works differ in the kind of program synthesized and the techniques used. The only work that uses a constraint-based approach is that of Colón, who synthesizes

imperative programs computing polynomial functions from partially specified programs and their invariants [5].

8 Conclusion

This paper formalized the notion of inductive controlled invariants and showed that inductive controlled invariants can be used to synthesize controllers that satisfy some safety requirements. Theoretically, this approach is sound and complete. We adapted this approach to the problem of synthesizing switching logic for multi-modal systems. We presented several sufficient conditions for a set to be an inductive controlled invariant set for a multi-modal dynamical system. These sufficient conditions were used to synthesize controllers using template-based techniques, which were then adapted to generate optimal controlled invariants.

References

[1] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(3), 3–34 (1995)
[2] Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. Proc. IEEE 88(7), 1011–1025 (2000)
[3] Blanchini, F.: Set invariance in control. Automatica 35, 1747–1767 (1999)
[4] Burns, K., Gidea, M.: Differential Geometry and Topology: With a view to dynamical systems. Chapman & Hall, Boca Raton (2005)
[5] Colón, M.: Schema-guided synthesis of imperative programs by constraint solving. In: LOPSTR, pp. 166–181 (2004)
[6] Cury, J., Brogh, B., Niinomi, T.: Supervisory controllers for hybrid systems based on approximating automata. IEEE Trans. Aut. Control 43, 564–568 (1998)
[7] Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
[8] Koo, T., Sastry, S.: Mode switching synthesis for reachability specification. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 333–346. Springer, Heidelberg (2001)
[9] Manon, P., Valentin-Roubinet, C.: Controller synthesis for hybrid systems with linear vector fields. In: Proc. IEEE Symp. on Intell. Control, pp. 17–22 (1999)
[10] Moor, T., Raisch, J.: Discrete control of switched linear systems. In: Proc. Eur. Control Conf. ECC 1999 (1999)
[11] Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
[12] Sankaranarayanan, S., Sipma, H., Manna, Z.: Constructing invariants for hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 539–554. Springer, Heidelberg (2004)
[13] Tomlin, C., Lygeros, L., Sastry, S.: A game-theoretic approach to controller design for hybrid systems. Proc. of the IEEE 88(7), 949–970 (2000)

Extending Symmetry Reduction by Exploiting System Architecture

Richard Trefler[1,⋆] and Thomas Wahl[2,⋆⋆]

[1] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[2] Computer Systems Institute, ETH Zurich, Switzerland
and
Oxford University Computing Laboratory, Oxford, United Kingdom

Abstract. Symmetry reduction is a technique to alleviate state explosion in model checking by replacing a model of replicated processes with a bisimilar quotient model. The size of the quotient depends strongly on the set of applicable symmetries, which in many practical cases allows only polynomial reduction. We introduce *architectural symmetry*, a concept that exploits architectural system features to compensate for a lack of symmetry in the system model. We show that the standard symmetry quotient of an architecturally symmetric and *well-architected* model preserves arbitrary Boolean combinations and nestings of reachability properties. This quotient can be exponentially smaller than the model, even in cases where traditional symmetry reduction is nearly ineffective. Our technique thus extends the benefits of symmetry reduction to systems that are in fact not symmetric. Finally, we generalize our results to all architecturally symmetric models, including those that are not well-architected. We illustrate our method through examples and experimental data.

1 Introduction

Symmetry is a feature of many multi-process systems that can be exploited in order to alleviate state explosion in model checking. A symmetry is a permutation of process indices that leaves the system model invariant. The idea of symmetry reduction is to replace the model by a smaller and bisimilar *quotient* that contains only one of the many states from the original model that are identical up to permutations. When the set of symmetries is large, the quotient model can be significantly smaller than the model under verification. In particular, under *full* symmetry the system model is invariant under all permutations of the process indices. This scenario is attractive for symmetry reduction as it allows an exponential reduction in model size.

Unfortunately, a system model may not be fully symmetric even when the system consists of replicated processes that execute the same parametrized program. Consider, for instance, protocols that assume a ring-like communication structure, in the style of the Dining-Philosophers resource allocation scheme.

⋆ Supported by an Individual Discovery grant from NSERC, Canada.
⋆⋆ Supported by the EPSRC, United Kingdom, grant no. EP/G026254/1.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 320–334, 2009.

The fact that a process may only communicate with its neighbor to the "right" breaks full symmetry, as neighborhood is not preserved by arbitrarily rearranging processes. Similarly, consider the cache coherence problem in a modern multi-core hardware design. As the number of cores continues to grow, they will likely not be pairwise connected, but instead exhibit lean communication topologies, permitting only few process symmetries. As a result, symmetry reduction of a significant magnitude cannot be expected.

In this paper we address this lack of reduction by generalizing existing symmetry-based techniques for multi-process programs. More precisely, we show that full symmetry reduction is applicable to systems that are *not* fully symmetric, provided (i) the system model is *architecturally symmetric*, (ii) the system model is *well-architected*, and (iii) the property of interest is expressible in a rich subset of CTL called *Safety CTL*. Under these conditions, which we explain below, *architectural symmetry reduction* produces a quotient structure that is an *exact* abstraction and can be exponentially smaller than the traditional symmetry quotient.

Conditions (i) through (iii) form the core of our approach. An architectural symmetry of a structure M is a permutation of the process indices that leaves the *positive transitive closure* of M's transition relation R invariant, i.e. M's reachability relation R^+. In contrast, traditional symmetry reduction requires more strictly that R itself be invariant under permutations. A system model is well-architected if it provides, at all reachable system states, the possibility of return to an initial state. This property is common in reactive protocols that continuously respond to user requests, and in many non-terminating systems as a means to counter the effects of resource leaks. Finally, Safety CTL consists of all formulas built out of Boolean connectives and CTL's EF operator, including arbitrary nestings. We show that the *standard* symmetry quotient of a well-architected and architecturally symmetric system preserves—in both directions—Safety CTL properties. We formalize this preservation property in the notion of *safety bisimulation*, a relationship between structures that allows transitions in one system to be simulated by finite-length paths in the other and is therefore weaker than traditional bisimulation.

In summary, the contribution of this paper is to extend a well-known and popular technique, symmetry reduction, to a class of systems that the technique was previously considered inapplicable to. Given conditions (i) through (iii), any existing symmetry reduction technique can be applied to the model—no new reduction algorithm is required. This makes our technique combine effortlessly with existing tools. We finally extend our results to programs that are not well-architected: we show that architectural symmetry alone is enough to give rise to an exponentially smaller quotient that preserves—again in both directions—reachability properties.

2 Background

2.1 Kripke Structures

We use the standard nomenclature. Let AP be a finite set of *atomic propositions*. A *Kripke structure* is a tuple $M = (S, R, L, I)$, where S is a finite set of *states*,

$R \subseteq S \times S$ is a set of *transitions* (state changes) of M, $L\colon S \to 2^{AP}$ is a *labeling function* that assigns to each state a set of atomic propositions considered *true* in that state, and finally $I \subseteq S$ is a set of designated initial states of M. A *path in M from s to t* is a sequence $(p^i)_{i=0}^k$ of states such that $k \geq 0$, $p^0 = s$, $p^k = t$ and $(p^i, p^{i+1}) \in R$ for all i with $0 \leq i < k$. The *length* of a path is the number of transitions in it. For instance, path $(p^i)_{i=0}^k$ has length k. A state t is *reachable* in M if there is a path in M from some initial state to t. We write R^+ for the positive transitive closure of R, i.e. R^+ is the smallest set such that

1. $R \subseteq R^+$, and
2. whenever $(u, v) \in R^+$ and $(v, w) \in R^+$, then also $(u, w) \in R^+$.

We have $(s, t) \in R^+$ exactly if there is a path in M from s to t of length at least 1.

Bisimulation. Let $M_1 = (S_1, R_1, L_1, I_1)$ and $M_2 = (S_2, R_2, L_2, I_2)$ be Kripke structures over AP. A relation $\approx \subseteq S_1 \times S_2$ is a *bisimulation* if $s_1 \approx s_2$ implies:

1. $L_1(s_1) = L_2(s_2)$,
2. for every $t_1 \in S_1$ such that $(s_1, t_1) \in R_1$, there exists $t_2 \in S_2$ such that $t_1 \approx t_2$ and $(s_2, t_2) \in R_2$, and
3. for every $t_2 \in S_2$ such that $(s_2, t_2) \in R_2$, there exists $t_1 \in S_1$ such that $t_1 \approx t_2$ and $(s_1, t_1) \in R_1$.

If $\approx$ is a bisimulation, and for each $s_1 \in I_1$ there exists $s_2 \in I_2$ such that $s_1 \approx s_2$, and for each $s_2 \in I_2$ there exists $s_1 \in I_1$ such that $s_1 \approx s_2$, then M_1 and M_2 are *bisimilar*. Bisimilarity implies that the structures satisfy the same properties of the temporal logic *CTL*. CTL is the smallest set of formulas that comprises *false*, *true*, the atomic propositions (AP), and is closed under Boolean connectives and the *temporal modalities* EX, AX, EF, EG, EU, etc.

Canonical Quotients. Many existential abstractions are based on the formation of a *canonical* quotient of the given Kripke structure, as follows. Let $M = (S, R, L, I)$ and $\equiv$ be an equivalence relation on S such that $s \equiv t$ implies $L(s) = L(t)$, with equivalence classes written as $[s]$. The canonical quotient of M is given by the structure $M' = (S', R', L', I')$ such that

$$\begin{aligned} S' &= \{\, [s] : s \in S \,\}, \\ R' &= \{\, ([s], [t]) \in S' \times S' :\ \exists s_0 \in [s],\, t_0 \in [t] :\ (s_0, t_0) \in R \,\}, \\ L'([s]) &= L(s), \text{ and} \\ I' &= \{[s] : s \in I\}\,. \end{aligned}$$

The requirement that $s \equiv t$ imply $L(s) = L(t)$ ensures that L' is well-defined. As an example, let $\equiv_L$ be the *labeling equivalence* with respect to L, i.e. the relation on S defined by $s \equiv_L t$ iff $L(s) = L(t)$. Relation $\equiv_L$ is the *coarsest* equivalence relation that allows L' to be well-defined.

2.2 Symmetry in Multi-process Systems

The term "process" is used in this paper generically for a component of a concurrent system. A *state* $(\vec{g}, l_1, \ldots, l_n)$ in such a system consists of the values $\vec{g}$

of all global variables (not associated with any process) and the *local state* l_i of each process $i \in \{1, \ldots, n\}$ (values of all local variables of process i).

Symmetries of a Kripke model M are defined with respect to permutations (bijections) $\pi\colon S \to S$ on the state space S; we describe such permutations in more detail in the next paragraph. We extend π to a mapping $\pi\colon R \to R$ on the transition level by defining $\pi((s,t)) = (\pi(s), \pi(t))$.

Definition 1. *A permutation π on S is said to be a **symmetry** of Kripke structure $M = (S, R, L, I)$ if*

1. *R is invariant under π: $\pi(R) = R$, and*
2. *L is invariant under π: $L(s) = L(\pi(s))$ for any $s \in S$, and*
3. *I is invariant under π: $\pi(I) = I$.*

*The symmetries of M form a group under function composition. Model M is said to be **symmetric** if its symmetry group G is non-trivial; we speak of symmetry **with respect to** G.*

For an n-process system, a symmetry π is derived from a permutation on $\{1, \ldots, n\}$ and acts on a state $s = (\vec{g}, l_1, \ldots, l_n)$ as $\pi(s) = (\vec{g}^{\,\pi}, l_{\pi(1)}, \ldots, l_{\pi(n)})$. That is, the local states of the processes are permuted by permuting their positions in the state vector. Further, π acts on $\vec{g}$ by acting component-wise on each global variable g. The action of π on g depends on the nature of g; we refer the reader to [9] for details.

Exploiting symmetry. Given a group G of symmetries, the relation $s \equiv_o t$ iff $\exists \pi \in G : \pi(s) = t$ defines an equivalence between states and is known as the *orbit relation*; the equivalence classes it entails are called *orbits* [5], written $[s]$ for $s \in S$. Observe that $s \equiv_o t$ implies $L(s) = L(t)$, since L is invariant under permutations in G (definition 1). Let therefore $\overline{M}$ be the canonical quotient of M with respect to $\equiv_o$. Quotient $\overline{M}$ turns out to be bisimilar to M [5]. As a result, for two states $s \in S$, $\overline{s} \in \overline{S}$ with $s \in \overline{s}$ and any CTL formula f over AP whose atomic propositions are invariant under permutations in G,

$$M, s \models f \quad \text{iff} \quad \overline{M}, \overline{s} \models f. \tag{1}$$

Depending on the size of G, $\overline{M}$ can be up to exponentially smaller than M. For example, for full symmetry in n-process systems, all $n!$ many permutations of a global state with pairwise distinct local states are orbit-equivalent and can be collapsed into a single abstract state.

3 Safety Bisimulation

The goal of this paper is to dramatically reduce the verification complexity for certain systems with only little symmetry. The quotients that we obtain can therefore not be expected to be bisimilar to the original system model. Instead, we will use the following weaker notion.

Definition 2. *Let $M_1 = (S_1, R_1, L_1, I_1)$ and $M_2 = (S_2, R_2, L_2, I_2)$ be Kripke structures over AP. Relation $\approx_r \subseteq S_1 \times S_2$ is a **safety bisimulation** if $s_1 \approx_r s_2$ implies:*

1. $L_1(s_1) = L_2(s_2)$,
2. *for every* $t_1 \in S_1$ *such that* $(s_1, t_1) \in R_1$, *there exists* $t_2 \in S_2$ *such that* $t_1 \approx_r t_2$ *and* $(s_2, t_2) \in R_2^+$, *and*
3. *for every* $t_2 \in S_2$ *such that* $(s_2, t_2) \in R_2$, *there exists* $t_1 \in S_1$ *such that* $t_1 \approx_r t_2$ *and* $(s_1, t_1) \in R_1^+$.

If $\approx_r$ *is a safety bisimulation, and for each* $s_1 \in I_1$ *there exists* $s_2 \in I_2$ *such that* $s_1 \approx_r s_2$, *and for each* $s_2 \in I_2$ *there exists* $s_1 \in I_1$ *such that* $s_1 \approx_r s_2$, *then* M_1 *and* M_2 *are* ***safety-bisimilar***.

Safety bisimilarity is identical to bisimilarity except for the occurrences of R_2^+ and R_1^+ in conditions 2 and 3. Figure 1 shows pairs of safety-bisimilar structures that are not bisimilar. The safety bisimulation relates states with identical labels.

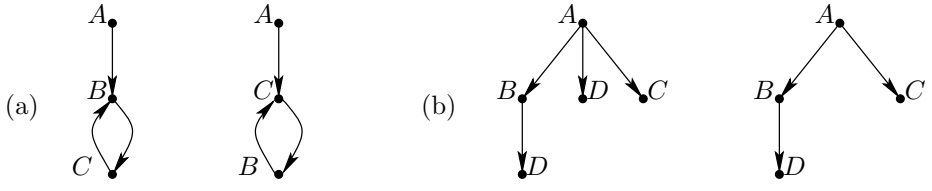


Fig. 1. Examples of safety-bisimilar structures

As with bisimilarity and CTL, there is a temporal logic that cannot distinguish safety-bisimilar structures.

Definition 3. ***Safety CTL****, denoted by EF-CTL, is the smallest set of formulas satisfying the following conditions:*

base formulas: *The Boolean constants false and true are EF-CTL formulas. For* $P \in AP$, P *is an EF-CTL formula.*
closure under Boolean connectives: *If* f *is an EF-CTL formula, so is* $\neg f$. *If* g *and* h *are EF-CTL formulas, so are* $g \wedge h$, $g \vee h$, *etc.*
closure under EF**:** *If* f *is an EF-CTL formula, so is* $\mathsf{EF}\, f$.

By definition, EF-CTL is a strict subset of CTL: neither *next-time* nor *until* operators can be expressed in (or generally translated into) EF-CTL. On the other hand, as common in CTL we use $\mathsf{AG}\, f$ as an abbreviation for $\neg\, \mathsf{EF} \neg f$. Thus, a CTL formula belongs to EF-CTL if any modality occurring in it is EF or AG. For instance, consider a system with two processes i and j and the property that it always be possible to reach a state in which the processes are synchronized. This property, which is neither of the classical safety nor liveness type, is expressed in EF-CTL as $\mathsf{AG}\, \mathsf{EF}\, \mathit{synch}(i, j)$.

Since EF-CTL is a sub-logic of CTL, we can define its semantics by resorting to CTL. We write $M, s \models f$ to mean that the EF-CTL formula f evaluates to *true* over structure M and state s, which in turn is to mean that with CTL semantics, $M, s \models f$.

We now establish the relationship between safety bisimilarity and EF-CTL:

Theorem 4. *Let $M_1 = (S_1, R_1, L_1, I_1)$ and $M_2 = (S_2, R_2, L_2, I_2)$ be Kripke structures over AP and $\approx_r$ a safety bisimulation between them. Let further s_1, s_2 be states with $s_1 \approx_r s_2$ and f be an EF-CTL formula. Then $M_1, s_1 \models f$ exactly if $M_2, s_2 \models f$.*

Proof. We show the $\Rightarrow$ direction; the inverse direction follows since the inverse relation $\approx_r^{-1} \subseteq S_2 \times S_1$ is a safety bisimulation as well. The proof is by induction on the structure of f.

(base formulas)
The interpretations of *false* and *true* are independent of M_1, s_1, M_2, s_2. Let $f \in AP$. From $s_1 \approx_r s_2$, it follows that $L_1(s_1) = L_2(s_2)$ and therefore $M_1, s_1 \models f$ implies $f \in L_1(s_1)$, hence $f \in L_2(s_2)$ and thus $M_2, s_2 \models f$.

(closure under Boolean connectives)
The result follows immediately from the induction hypothesis and the semantics of $\neg$, $\wedge$, $\vee$.

(closure under EF)
Suppose $M_1, s_1 \models \mathsf{EF}\, g$. This means that there is a path $p := (p^i)_{i=0}^k$ in M_1 with $p^0 = s_1$ and $M_1, p^k \models g$. We now claim that there exists a state $q \in S_2$ that is reachable from s_2 and satisfies $p^k \approx_r q$. Given this claim and $M_1, p^k \models g$, we apply the induction hypothesis to conclude that $M_2, q \models g$. Since q is reachable from s_2, this proves $M_2, s_2 \models \mathsf{EF}\, g$.

To show the claim, we proceed by induction on k. If $k = 0$, then $p^k = p^0 = s_1$. Choosing $q := s_2$ satisfies all requirements.

Assume now p has the form $(p^i)_{i=0}^{k+1}$, and consider the prefix $(p^i)_{i=0}^k$ of p. By the induction hypothesis (of the claim), there exists a state $q' \in S_2$ that is reachable from s_2 and satisfies $p^k \approx_r q'$. Further, $(p^k, p^{k+1}) \in R_1$. Since $\approx_r$ is a safety bisimulation between M_1 and M_2, there is a state $q \in S_2$ with $(q', q) \in R_2^+$ and $p^{k+1} \approx_r q$. In particular, q is reachable from q' and thus reachable from s_2 and satisfies all requirements of the claim. □

Figure 1 demonstrates that the addition of CTL's *next-time* (X) or *until* (U) operators is enough to distinguish safety-bisimilar structures. The CTL formula $\mathsf{EX}\, B$ is true of the first structure in (a), but not of the second. Likewise, the CTL formula $\mathsf{E}((\mathsf{EF}\, C)\, \mathsf{U}\, D)$ is true of the first structure in (b), but not of the second. In contrast, the two structures in (a) satisfy the same EF-CTL formulas, as do the two structures in (b).

4 Architectural Symmetry

We are now ready to define architectural symmetry and show that, under certain conditions, it permits a safety-bisimilar quotient. Looking back at definition 1 (symmetry), the crucial property of a symmetric model is its invariance under permutations. Since safety bisimilarity is weaker than bisimilarity, we can afford a weaker invariance notion, thus capturing a larger class of systems.

Definition 5. *A permutation π on S is said to be an* ***architectural symmetry*** *of the Kripke structure $M = (S, R, L, I)$ if*

1. R^+ *is invariant under* π*:* $\pi(R^+) = R^+$,
2. L *is invariant under* π*: for any* $s \in S$, $L(s) = L(\pi(s))$, *and*
3. I *is invariant under* π*:* $\pi(I) = I$.

The architectural symmetries of M form a group under function composition. Structure M is said to be architecturally symmetric *if its architectural symmetry group G is non-trivial; we speak of architectural symmetry* with respect to G.

Note the difference to definition 1: in item 1, instead of requiring R to be permutation invariant, we require R^+ to be. Consider a finite path p between two states s and t. Under symmetry, the permuted sequence $\pi(p)$ is a valid path as well, connecting $\pi(s)$ and $\pi(t)$. Under architectural symmetry, all we can say is that $\pi(s)$ and $\pi(t)$ are connected in M as well, since $(\pi(s), \pi(t)) \in \pi(R^+) = R^+$.

Comparing the two types of symmetry, we confirm that architectural symmetry is weaker than symmetry:

Lemma 6. *If M is symmetric with respect to a group G, then M is architecturally symmetric with respect to G.*

Proof. We have to show that each symmetry is an architectural symmetry. Let $\pi \in G$ be a permutation on S. Requirements 2 and 3 are identical in definitions 1 and 5. Regarding requirement 1, suppose $\pi(R) = R$, we show $\pi(R^+) \subseteq R^+$ (the other inclusion follows with a symmetric argument; note that R^+ is a finite set).

To this end, consider $(s_1, t_1) \in \pi(R^+)$, i.e. $(s_1, t_1) = \pi((s_2, t_2))$ for some pair $(s_2, t_2) \in R^+$. Let p_2 be a path in M that connects s_2 to t_2. Each transition of p_2 belongs to R. Therefore, each transition of $p_1 := \pi(p_2)$ belongs to $\pi(R) = R$, so p_1 is a valid path. Since p_1 connects $\pi(s_2) = s_1$ to $\pi(t_2) = t_1$, it follows that $(s_1, t_1) \in R^+$. □

Example. In the following we demonstrate that lemma 6 can in general not be strengthened is not an equivalence. Consider a token ring model where the shared token regulates access to some resource. Such rings occur in hardware models and in communication protocols. Figure 2 shows the local transition diagram of process i. The process may be in one of the local states N, N^+, T, T^+, C; there are no global variables. Intuitively, N, T and C indicate that the process is "not trying" to access the resource, "trying" to do so, or is "currently" accessing it. The superscript $^+$ indicates ownership of the token. The process can move freely between local states N and T, and also between N^+ and T^+. To acquire the token, it must currently be possessed by the left neighbor, process $i-1$ ($i-1$ and $i+1$ are defined cyclically within the index range $\{1, \ldots, n\}$), and that neighbor must be willing to release the token. This is indicated by the simultaneous transition $N^+_{i-1} \to N_{i-1}$ in figure 2. Analogously, to release the token it must be received by process $i+1$, which must be ready to do so (indicated by N_{i+1} or T_{i+1}). Let finally

$$I \quad := \quad \{(s_1, \ldots, s_n) : \exists i : s_i = N^+ \ \wedge \ \forall j : j \neq i : s_j = N\}$$

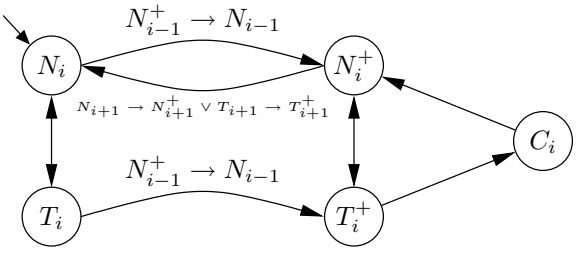


Fig. 2. Token ring example for resource allocation lacking full symmetry

be the set of initial states: every process is non-trying, and any one of them owns the token.

It is easy to prove that the Kripke structure M induced by the parallel composition of n processes running the program in figure 2 enjoys rotational symmetry: permutations of the cycle form $(1\ 2\ \ldots\ n)$ leave the structure invariant. M is not, however, fully symmetric: consider $n = 3$ and the transition

$$\tau \quad := \quad (N_1^+,\ T_2,\ T_3) \quad \to \quad (N_1,\ T_2^+,\ T_3)\,.$$

Applying the transposition (1 2) to τ results in the two states $(T_1,\ N_2^+,\ T_3)$ and $(T_1^+,\ N_2,\ T_3)$. The transition from T_1 to T_1^+ is not allowed by figure 2 in the context of state $(T_1,\ N_2^+,\ T_3)$. In fact, consider any permutation π such that $\pi(i-1)+1 \neq \pi(i)$. Then π is not a symmetry of M, by the same argument. As a consequence, for a symmetry π the condition $\pi(i-1)+1 = \pi(i)$ is necessary for all i, and it is also sufficient. Thus, the rotation group is the largest symmetry group of M, and one cannot expect more than linear savings due to standard symmetry reduction for this structure.

On the other hand, applying definition 5, we see that M is *fully* architecturally symmetric. We first show that $\pi(R) \subseteq R^+$. The only interesting cases are the transitions where process i acquires the token from its left neighbor $i-1$. After permuting such a transition, the process $\pi(i-1)$ releasing the token is generally someone other than the left neighbor. The resulting invalid transition can be simulated by a path that passes the token from $\pi(i-1)$ successively to that process' right neighbors until it eventually reaches process $\pi(i)$ (some temporary moves from T to N may be required to enable the passing of the token). To show that $\pi(R^+) = R^+$, one applies this idea to each transition of a permuted path and connects the resulting paths to a (long) final path. Also, I is invariant under arbitrary permutations, and L can be defined to be so. In conclusion, M features an exponential-size architectural symmetry group, but only a small polynomial-size standard symmetry group. □

Before we demonstrate the benefits of reducing architecturally symmetric systems in the next section, we show the following property.

Lemma 7. *Let M be architecturally symmetric with respect to G and Reached be the set of reachable states of M. Then, for all $\pi \in G$, $\pi(Reached) = Reached$.*

Proof. We show $\pi(\mathit{Reached}) \subseteq \mathit{Reached}$ (the other inclusion follows with a symmetric argument; note that *Reached* is a finite set). Assume $\pi(R^+) = R^+$, and consider $t \in \pi(\mathit{Reached})$, i.e. $t = \pi(r)$ for some $r \in \mathit{Reached}$. Then there is some initial state $s \in I$ such that $(s, r) \in R^+$, i.e. $\pi(s, r) = (\pi(s), \pi(r)) = (s', t) \in \pi(R^+) = R^+$, where $s' := \pi(s)$. Since, by definition 5, I is invariant under π, it follows that $s' \in I$. Thus, t is reachable in M (namely, from s'), i.e. $t \in \mathit{Reached}$. □

5 Well-Architected Systems

Architectural symmetry alone is not yet enough to permit a safety-bisimilar quotient. We therefore now consider models satisfying the following condition:

Definition 8. *A system model $M = (S, R, L, I)$ is* ***well-architected*** *if*

1. *M's initial states are all reachable from each other, and*
2. *for every reachable state s, there is an initial state that is reachable from s.*

The possibility of returning to the initial state at any time is common in reactive systems to prevent resource leaks in long-running executions, for instance through *micro-reboots* [3]. Communication protocols in the IP and telephony communities regulate the coexistence of interacting *features*. Each feature is a finite-state terminating process; overall behavior is described by continuously selecting an appropriate feature based on current input, executing the feature to completion, issuing appropriate output and then returning to some (often the unique) initial state.

As a concrete example, the model of the resource allocation scheme shown in figure 2 is well-architected: first, the initial states are reachable from each other, since the token can be passed around until it reaches which ever process requested to have it. Second, *every* initial state can be reached from any reachable state by letting each process individually return to its initial local state (N or N^+); the token may again have to be passed around to whoever held it initially.

From the definition of well-architectedness, we conclude:

Observation 9. *Let $M = (S, R, L, I)$ be well-architected and u and v be reachable states. Then u and v are reachable from each other.*

Proof. Since M is well-architected, there is a path from u to some $s \in I$. Since v is reachable, there is a path from some $t \in I$ to v. Again since M is well-architected, s and t are mutually reachable. Putting it all together, there is a path from u to v. The reachability of u from v follows symmetrically. □

Architectural Symmetry Quotients of Well-Architected Systems
We now present the main result of this paper: From a well-architected and architecturally symmetric model M, one can derive a safety-bisimilar quotient structure M'. Quotient M' is obtained as the canonical quotient (see section 2.1) of M with respect to the orbit relation $\equiv_o$ on S. In other words, EF-CTL formulas can be verified reliably over the standard symmetry quotient, although the underlying model is *not* symmetric.

Theorem 10. *Let M be well-architected and architecturally symmetric with respect to G. Let further $\equiv_o$ be the orbit relation on S, i.e. $s \equiv_o t$ iff $\exists \pi \in G : \pi(s) = t$. Let finally M' be the canonical quotient of structure M with respect to $\equiv_o$. Then M' is safety-bisimilar to M.*

Proof. We first remark that the labeling function of the quotient is well-defined: Let $s \equiv_o t$. Then there exists $\pi \in G$ such that $\pi(s) = t$. Since M is architecturally symmetric, L is invariant under π, which implies $L(s) = L(t)$.

We now define a suitable relation $\equiv_r$ between S and S', namely:

$$s_1 \equiv_r [s_2] \quad \text{iff} \quad s_1 \equiv_o s_2 \,.$$

In particular, $s \equiv_r [s]$ for any $s \in S$. We claim that $\equiv_r$ is a safety bisimulation. The theorem then follows, since the initial states of M and M' are appropriately related: for any $s \in I$, it is $s \equiv_r [s] \in I'$. Further, for any $[s] \in I'$, it is $[s] \equiv_r s \in I$. To show the claim, let $s_1 \equiv_r [s_2]$, hence $s_1 \equiv_o s_2$ and thus $[s_1] = [s_2]$. We prove the three conditions of definition 2. We restrict our attention to the reachable part of M, which is commonly achieved by exploring M and building the quotient on the fly. That is, s_1 is an actually reached and, therefore, reachable state of M.

1. By the remark above about $\equiv_o$, we obtain $L(s_1) = L(s_2)$, and by the definition of M', $L(s_2) = L'([s_2])$. Thus $L(s_1) = L'([s_2])$.
2. Let t_1 be such that $(s_1, t_1) \in R$. We choose $t_2 := t_1$ and consider $[t_2]$: It is $t_1 \equiv_r [t_1] = [t_2]$. Further, from $(s_1, t_1) \in R$ we conclude $([s_1], [t_1]) = ([s_2], [t_2]) \in R' \subseteq R'^+$.
3. Let $[t_2]$ be such that $([s_2], [t_2]) \in R'$. By definition of R', there exist $s \in [s_2]$, $t \in [t_2]$ such that $(s, t) \in R$. We conclude $t \equiv_r [t_2]$. Further, from $s_1 \equiv_o s_2$ and $s \equiv_o s_2$, we conclude $s_1 \equiv_o s$. By lemma 7 and the reachability of s_1, it follows that s is also reachable in M. Therefore t is reachable in M. Since s_1 is also reachable in M, by observation 9 there is a path from s_1 to t, which implies $(s_1, t) \in R^+$. We now choose $t_1 := t$ to obtain $t_1 = t \equiv_r [t_2]$ and $(s_1, t_1) = (s_1, t) \in R^+$.

Note that well-architectedness was used only in the form of observation 9. □

We summarize this section in the following statement:

Corollary 11. *M and M' as in theorem 10 satisfy the same EF-CTL properties.*

We emphasize again that, assuming G is the full symmetry group, M' is exponentially smaller than M, although standard symmetry may allow only an insignificant reduction. Note, however, that in order to apply full architectural symmetry reduction, the EF-CTL properties of interest must have fully symmetric atomic propositions.

Considering again the example in figure 2, which allows only polynomial symmetry reduction: Since it is both well-architected and architecturally symmetric with respect to the full symmetry group, we can apply full symmetry reduction to it when verifying EF-CTL formulas, giving rise to an exponentially smaller quotient. In section 7 we underpin this result with quantitative data.

6 Generalization: Non-Well-Architected Systems

We briefly demonstrate that well-architectedness is not even required if one restricts the set of eligible formulas further, namely to reachability properties:

Theorem 12. *Let $M = (S, R, L, I)$ be architecturally symmetric with respect to group G, and let M' be the canonical quotient of M with respect to the orbit relation. For any state $s \in S$ and an atomic proposition q,*

$$M, s \models \mathsf{EF}\, q \quad \textit{iff} \quad M', [s] \models \mathsf{EF}\, q\,.$$

Analogously, the theorem can be stated as $M, s \models \mathsf{AG}\, q$ iff $M', [s] \models \mathsf{AG}\, q$. In other words, for architecturally symmetric systems, safety properties such as the unreachability of an error state can be equivalently formulated over the quotient M'. To prove the theorem, we first show a stronger *path correspondence* result. It addresses the "disconnect problem" of existential abstractions, namely that in general a path in the abstract system may not be liftable to one in the concrete system. It turns out that under architectural symmetry, it is.

Lemma 13. *Let M and M' be as above and $(p'^i)_{i=0}^k$ be a path in M'. Then, for any $s \in p'^0$, there is a path in M from s to some element $t \in p'^k$.*

Proof. By induction on k. If $k = 0$, choose $t := s$ to get a path in M of length 0. Now consider path $(p'^i)_{i=0}^{k+1}$, and let $s \in p'^0$. By the induction hypothesis, there is a path p in M from s to some state $t^k \in p'^k$. Further, $(p'^k, p'^{k+1}) \in R'$ implies that there is a transition $(x, y) \in R$ such that $x \in p'^k$, $y \in p'^{k+1}$. Then $x \equiv_o t^k$, so let $\pi \in G$ be a permutation such that $\pi(x) = t^k$. By architectural symmetry, $(x, y) \in R \subseteq R^+ = \pi(R^+)$, thus $(\pi(x), \pi(y)) = (t^k, \pi(y)) \in R^+$. Concatenating path p and the path from t^k to $\pi(y)$ results in a path in M from s to $t^{k+1} := \pi(y) \in p'^{k+1}$. □

Proof. [Theorem 12]:

"⇒": Any path in M from s to t satisfying q can be mapped to a path in M' from $[s]$ to $[t]$. By the definition of L', $q \in L(t) = L'([t])$.

"⇐": Suppose $M', [s] \models \mathsf{EF}\, q$, i.e. there is a path p' in M' with $p'^0 = [s]$ and $q \in L'(p'^k)$ for some k. By lemma 13, since $s \in [s] = p'^0$, there is a path in M from s to some element $t \in p'^k$. Thus, $q \in L'(p'^k) = L(t)$ by requirement 2 of definition 5, proving $M, s \models \mathsf{EF}\, q$. □

7 Experiments and Further Examples

In this section we present some quantitative data to support our proposed technique. We consider the token ring example from section 4 and show the difference between model checking this system by exploiting standard symmetry, and by exploiting architectural symmetry. We have already established that the Kripke structure induced by the system is rotationally symmetric, and that it is also both well-architected and architecturally symmetric with respect to the full symmetry group.

Table 1. Space and time requirements for the token ring example

		Rotational Symmetry		Architectural Symmetry	
Numb. of processes	BDD nodes Trans. Rel.	Numb. of BDD nodes	Time	Numb. of BDD nodes	Time
40	1,647	46,103	0:07m	5,612	0:02m
50	2,067	70,448	0:23m	7,052	0:06m
60	2,487	99,893	0:53m	8,492	0:11m
70	2,907	134,438	1:36m	9,932	0:21m
80	3,327	174,083	2:48m	11,372	0:35m
90	3,747	218,828	4:33m	12,812	0:58m
100	4,167	268,673	6:59m	14,252	1:24m

We conducted experiments using the SVISS symbolic verifier [2], an experimental platform for symmetric systems. SVISS is based on the CUDD BDD library [16] and supports various symmetry groups, in particular the rotational and the full group, which are relevant for our example. We ran the example on a 2GB main memory dual-core 2.2GHz system. The property we verified is mutually exclusive occupancy of the C local state. This property is satisfied on this system, so that SVISS generates the full reachable state space (up to symmetry reduction).

The table shows, for a growing number of processes executing the protocol, the size of the BDD for the transition relation, the maximum number of live BDD nodes ("Numb. of BDD nodes") during the verification run, and the running time. This example, although small, does impart the difference an exponential reduction makes over a polynomial one, namely the potential to scale up to large examples, especially regarding memory, the classical bottleneck of BDDs.

Architectural symmetry and multi-core memory consistency. With the advent of multi-core hardware designs, pairwise connected communication topologies will be too costly to support. Instead, hardware and software communication topologies based on rings, tori, trees, hypercubes, and specially designed patterns will likely abound [1]. No matter what the exact topology, it will be necessary to ensure some form of data consistency among the various cores accessing a shared memory segment. That is, for a particular memory location accessed by several cores, and possibly several internal core-level caches, the values stored in those processor locations should be consistent.

Violation of multi-core memory consistency can be stated using a formula of the form $\mathsf{EF}(\exists i, j : v(p_i) \neq v(p_j))$, expressing the reachability of a state where two processors p_i, p_j have different values for a single memory location v. This formula has fully symmetric atomic propositions, so that architectural symmetry reduction techniques can be applied in a straightforward manner. As a consequence, the property can be verified over architecturally symmetric systems (whether well-architected or not), enjoying the same reduction as fully symmetric ones, namely with an exponentially smaller quotient.

FlexRay, Time-triggered architectures. In the automotive electronics industry, the *FlexRay* consortium has been formed by major car manufacturers to design a communication protocol for the control logic in vehicles [10]. Bus and star networks are supported, as well as any hybrid topology resulting from a combination of bus and stars. Many dozens of nodes can be connected in a FlexRay network, making full interconnection too expensive. Similar structures with little conventional symmetry are supported by the *time-triggered protocol*, where the network is a broadcast bus, often equipped with dual channels for fault tolerance [11].

8 Conclusion and Outlook

We have described a new notion of *architectural symmetry*, which extends attractive benefits of symmetry reduction to many systems with little symmetry. The result is a potential for an exponentially more effective reduction in model size. The price we pay is an architectural requirement of *well-architectedness* and a specification language with less expressive power than CTL, namely EF-CTL. We have given examples of multi-process systems that can be fully symmetry-reduced, although the model under verification is only rotationally symmetric. We have finally shown that the requirement of well-architectedness can be traded in for a restriction to reachability properties.

Relation to previous work. Symmetry reduction for model checking was introduced in [5,7], and in [12] using *scalarsets* for fully symmetric systems. These works demonstrate the potential of symmetry for an exponential reduction in system size. This potential can in practice be thwarted if the symmetry is only "approximate": some permutations in the targeted symmetry group do not leave the model invariant. The work of [8,6,18] generalized symmetry reduction to systems where, despite the imprecision in the symmetry, a bisimilar quotient can be constructed. The results in [15,17] allow in principle arbitrary deviations from symmetry, but the reduction of course dwindles with the divergence from perfect symmetry. Our work, in contrast, deals with a different reason for limited effect of symmetry reduction: an insignificant symmetry group. To the best of our knowledge, our work is the first to apply symmetry reduction based on a *large* (say, the full) group to a model featuring a *small* (say, the rotational) group.

Our notion of safety bisimulation bears some resemblance with that of *weak bisimulation* [13]. The latter relates systems that are bisimilar up to externally unobservable actions, often called τ-transitions. Our setting is in a sense lower-level, as we do not distinguish between visible and invisible system steps and thus do not have τ-transitions.

Unrelated to symmetry, [4] defines an *implementation* relation that compares sets of executions rather than computation trees. In addition, unlike our notions of safety simulation and *bi*simulation, that notion does not seem to generalize to equivalence of structures. In particular, it does not guarantee that if an abstract model fails to satisfy a property, then so does the concrete model.

Future work. We plan to investigate precisely how to detect well-architectedness and architectural symmetry. If a system model is well-architected, it is usually not so by coincidence, but *by design*. For such systems, suspected well-architectedness can perhaps be verified at a high-level abstraction layer, akin to symmetry being verified or enforced at the program text level. As a last resort, well-architectedness can also be verified at the structure level, using a reachability pass from I forward, resulting in a set *Reached*, and one from I backward, resulting in a set $Reached^{-1}$. The structure is well-architected exactly if $Reached \subseteq Reached^{-1}$. Contrast the cost of this check with verifying symmetry, which is *graph-isomorphism complete*.

Regarding architectural symmetry, our approach to detecting it is based on the observation sketched earlier that $\pi(R^+) = R^+$ iff $\pi(R^+) \subseteq R^+$ iff $\pi(R) \subseteq R^+$. That is, a model is architecturally symmetric exactly if every permuted transition can be simulated by a finite-length path.

An open question is how symmetry reduction based on *process counters* [8,14] can be applied to a system architecturally symmetric with respect to the full symmetry group. Since our approach does not require full symmetry, a translation of the program text as described in [9] is not quite applicable.

Acknowledgments. The authors wish to thank E. Allen Emerson for his inspirational comments on this work, and Georg Weissenbacher for suggesting practical motivations and for revisions on early drafts.

References

1. Asanovic, K., et al.: The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
2. Wahl, T., Blanc, N., Emerson, E.A.: SVISS: Symbolic verification of symmetric systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 459–462. Springer, Heidelberg (2008)
3. Candea, G., Cutler, J., Fox, A.: Improving availability with recursive microreboots: a soft-state system case study. Performance Evaluation (2004)
4. Chen, X., German, S., Gopalakrishnan, G.: Transaction based modeling and verification of hardware protocols. In: Formal Methods in Computer-Aided Design (FMCAD) (2007)
5. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. In: Formal Methods in System Design (FMSD) (1996)
6. Emerson, A., Havlicek, J., Trefler, R.: Virtual symmetry reduction. In: Logic in Computer Science (LICS) (2000)
7. Emerson, A., Sistla, P.: Symmetry and model checking. Formal Methods in System Design (FMSD) (1996)
8. Emerson, A., Trefler, R.: From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 142–157. Springer, Heidelberg (1999)
9. Emerson, A., Wahl, T.: On combining symmetry reduction and symbolic representation for efficient model checking. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 216–230. Springer, Heidelberg (2003)

10. The FlexRay Consortium, FlexRay—The communication system for advanced automotive control applications, http://www.flexray.com
11. Heiner, G., Thurner, T.: Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In: Fault-Tolerant Computing Symposium (FTCS) (1998)
12. Ip, N., Dill, D.: Better verification through symmetry. Formal Methods in System Design (FMSD) (1996)
13. Milner, R.: Operational and algebraic semantics of concurrent processes. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. MIT Press, Cambridge (1990)
14. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 107. Springer, Heidelberg (2002)
15. Sistla, P., Godefroid, P.: Symmetry and reduced symmetry in model checking. Transactions on Programming Languages and Systems (TOPLAS) (2004)
16. Somenzi, F.: The CU Decision Diagram Package, release 2.3.1. University of Colorado at Boulder, http://vlsi.colorado.edu/~fabio/CUDD/
17. Wahl, T.: Adaptive symmetry reduction. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 393–405. Springer, Heidelberg (2007)
18. Wei, O., Gurfinkel, A., Chechik, M.: Identification and counter abstraction for full virtual symmetry. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 285–300. Springer, Heidelberg (2005)

Shape-Value Abstraction for Verifying Linearizability

Viktor Vafeiadis

Microsoft Research, Cambridge, UK

Abstract. This paper presents a novel abstraction for heap-allocated data structures that keeps track of both their shape and their contents. By combining this abstraction with thread-local analysis and rely-guarantee reasoning, we can verify a collection of fine-grained blocking and non-blocking concurrent algorithms for an arbitrary (unbounded) number of threads. We prove that these algorithms are linearizable, namely equivalent (modulo termination) to their sequential counterparts.

1 Introduction

Linearizability [1] is the standard correctness criterion for high-performance libraries of concurrent data structures, such as `java.util.concurrent` and Intel's TBB (thread building blocks). Linearizability is a safety property. Informally, a library is *linearizable* if calling any of its exported operations appears to execute atomically at some instant between its invocation and its return. This instant when the entire observable effect of a method is deemed to occur is known as the *linearization point*. Equivalently, a concurrent library is linearizable if every concurrent execution consisting of calls to its exported operations is equivalent to a sequential execution that preserves the order of non-overlapping operations. Therefore, a linearizable library can be fully specified by its sequential interface; any interesting concurrency is hidden inside the library.

One can easily achieve this atomicity with global lock, but concurrency experts use multiple fine-grained locks and non-blocking instructions, such as compare and swap (CAS), to get better performance and scalability. However, even these experts make mistakes, and it is not unusual for published concurrent algorithms to have subtle errors. Our aim is to provide automated verification tools to these experts so that they can formally verify the correctness of their algorithms.

The literature contains several hand-crafted linearizability proofs [2,3,4,5], but until recently nobody had automated the derivation of such proofs. Amit et al. [6] used shape analysis to verify linearizability for a fixed (small) number of threads. More recently, Manevich et al. [7] and Berdine et al. [8] extended this analysis, so that it works for a larger (fixed) number of threads and for an unbounded number of threads respectively. These works require a specialized abstract domain, do not handle memory deallocation, and do not prove that the linearization point occurred exactly once for each method call.

In contrast, we check that that the specified linearization points are sound, and we allow complex linearization points that occur in a different thread than the one being verified.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 335–348, 2009.

Our prototype implementation is based on RGSep [9], a program logic that combines rely-guarantee [10] and separation logic [11]. As a result, it deals with an unbounded number of threads, can reason about memory deallocation, which affects linearizability in subtle ways (see Sect. 2), and can prove the absence of memory leaks (where applicable).

Main Results. The contributions of this paper are summarised below:

- We present a simple proof method for verifying linearizability given a specified set of linearization points (see Sect. 3). Our method can handle linearization points occuring in a different thread than the one being verified.
- Our shape analysis can remember an adjustable amount of information about the values stored in a data structure (see Sect. 4). The amount of information can be adjusted by selecting a different backend value abstraction.
- We replace the complex RGSep atomic proof rule with two rules, thereby simplifying the presentation and enabling concise actions specifications for operations such as CAS (see Sect. 5).
- Our tool compares favourably to the other known tools, and succeeded in proving that several concurrent algorithms are linearizable (see Sect. 6).

Limitations. (1) We assume a sequentially consistent memory model; this means that parallel composition can be understood as trace interleaving. (2) The program must be accurately analysable by (sequential) shape analysis: this currently restricts our analysis to programs operating on linked lists. (3) The programmer must annotate the locations of the linearization points. (4) The programmer must describe the interference imposed by the module.

2 A Simple Example: Treiber's Stack

Figure 1 contains C-like pseudocode for Treiber's stack [12], one of the simplest non-blocking concurrent algorithms. The stack is represented as a singly linked list rooted at `S->Top`, which is updated using CAS (compare and swap). CAS is a primitive operation that reads a word from a memory adress and conditionally writes to the same address in one atomic step. In particular, `CAS(&S->Top,t,x)` atomically compares the value of `S->Top` with the value of `t` and if the two match, the `CAS` succeeds: it stores the value of `x` in `S->Top` and returns `1`. Otherwise, the `CAS` fails: it returns `0` and does not change the value of `S->Top`.

This algorithm leaks memory: we cannot free popped nodes because of the following scenario. Assume the stack initially consists of the nodes α and β. First, thread T calls `pop`, executes lines `21`–`25` and is then descheduled. At this point, T's local state is `t` $= \alpha$ and `x` $= \beta$. Now, suppose some other thread comes along and pops α off the stack and then pushes γ onto the stack. If `pop` were to dispose node α, it is possible for a new node to be allocated at the same address α and pushed on the stack. Hence, the stack can reach a configuration consisting of the nodes α, γ, and β. If T is rescheduled at this point, the CAS at line `26` will succeed, but will remove two nodes from the stack instead of one. This is known as the 'ABA' problem in the literature.

```
                                [10] void push(value_t v) { struct node *t, *x;
struct node {                   [11]   x = alloc();
  struct node *next;            [12]   x->data = v;
  value_t data;                 [13]   do {
};                              [14]     t = S->Top;
                                [15]     x->next = t;
struct stack {                  [16]   } while (¬CAS(&S->Top,t,x));    // @1
  struct node *Top;             [17] }
};
                                [20] value_t pop() { struct node *t, *x;
struct stack *S;                [21]   do {
                                [22]     t = S->Top;                  // @2
void init() {                   [23]     if (t == NULL)
   S = alloc();                 [24]       return EMPTY;
   S->Top = NULL;               [25]     x = t->next;
/* ABS->val = ϵ; */            [26]   } while (¬CAS(&S->Top,t,x));    // @3
}                               [27]   return t->data;
                                [28] }
```

Fig. 1. Treiber's non-blocking stack algorithm

Linearization Points. The linearization points are annotated with comments at the right-hand side. All of them are conditional: `@1` and `@3` are linearization points if and only if the respective `CAS` succeeds; `@2` is a linearization point if and only if the value stored to `t` is `NULL`. (`@2` is the linearization point of a failed `pop` operation: at this point we know that the stack is empty.) To carry out the verification, we expect the programmer to annotate these points with auxiliary code asserting that they are linearization points.

Actions. In order to verify the given algorithm, we also require the user to specify a set of precondition-postcondition pairs (a.k.a. actions) that summarize the possible atomic effects of the algorithm. For Treiber's stack, we need:

$$\begin{array}{ll}
\texttt{action APush()} & [\texttt{S}\mapsto\texttt{Top:}n \qquad\qquad\qquad\qquad * \texttt{ ABS}\mapsto\texttt{val:}A] \\
& [\texttt{S}\mapsto\texttt{Top:}y * y\mapsto\texttt{data:}e\texttt{,next:}n * \texttt{ABS}\mapsto\texttt{val:}\langle e\rangle\cdot A] \\
\texttt{action APop()} & [\texttt{S}\mapsto\texttt{Top:}y * y\mapsto\texttt{data:}e\texttt{,next:}n * \texttt{ABS}\mapsto\texttt{val:}\langle e\rangle\cdot A] \\
& [\texttt{S}\mapsto\texttt{Top:}n * y\mapsto\texttt{data:}e\texttt{,next:}n * \texttt{ABS}\mapsto\texttt{val:}A]
\end{array}$$

These actions use separation logic notation[1] and (ignoring the `ABS` part) describe the effect of a successful `CAS` at lines `16` and `26` respectively. The italicized variables (e.g. n) are logical variables and are implicitly quantified over both assertions of an action. The other lines, as well as failed `CAS`es do not change any state visible to other threads.

`ABS` is an auxiliary variable representing the abstract stack that the algorithm supposedly implements. This is formalized as a mathematical sequence. We write

[1] $\texttt{S}\mapsto\texttt{Top:}n$ denotes that `S` is a pointer to a structure whose `Top` field contains n. The $*$ operator is similar to conjunction, but $P * Q$ also asserts that P and Q describe disjoint parts of the memory.

ϵ for the empty sequence, $\langle e \rangle$ for the singleton sequence consisting of e, and $\cdot$ for sequence concatenation. Action `APush` adds e to the beginning of the abstract stack. Conversely, `APop` removes e from the beginning of the abstract stack. If we initialise `ABS->val` to ϵ in the constructor `init()`, our tool is able to infer the following invariant:

$$J \stackrel{\text{def}}{=} \exists nv.\ \texttt{S}{\mapsto}\texttt{Top:}n * \mathsf{lseg}(n, \texttt{NULL}, v) * \texttt{ABS}{\mapsto}\texttt{val:}v,$$

which says that the concrete singly linked list represents the same value as is stored in the auxiliary variable `ABS`. (The predicate $\mathsf{lseg}(n, \texttt{NULL}, v)$ asserts that there is a singly list segment starting from n and ending with `NULL` that represents the sequence value v.) This invariant, also known as the *abstraction map*, is crucial for the linearizability proof, and is used as the precondition of `push` and `pop`.

3 Verifying Linearizability

Proving linearizability can be reduced to proving that one transition system simulates another transition system (e.g. [2,4]). The reduction is straightforward, but expensive: it converts a difficult problem into an even harder problem. Proofs done this way have involved significant human labour, especially in constructing the appropriate simulation relations between the two automata. In one case, Colvin et al. [4] even had to invent an intermediate automaton and construct two simulation relations.

Instead, we employ a simpler –but equally general– proof technique based on auxiliary code annotations. We assume that the programmer knows the linearization point of each method and he annotates this point in the source code. For simple algorithms, such as Treiber's stack, this task is straightforward and could perhaps be automated. More complicated algorithms generally require more annotations, but these are still manageable and, in any case, simpler than the corresponding simulation relations. For such examples, see [13, Chapter 5].

In order to prove that a method is linearizable, we need a specification describing the intended atomic effect of the method. In our examples, this specification is supplied by the user. If, however, the user does not provide such a specification explicitly, we can extract it from the code itself: we just symbolically execute the code in an isolated (sequential) environment. Usually this simplifies the source code quite dramatically. For example, the two `CAS`es in Fig. 1 always succeed in an isolated environment.

Hence, we can assume that the concrete program is annotated with its linearization points and its specification given as abstract code. To verify linearizability: we infer an abstraction map, J; we inline the specification at the annotated linearization points; and check the following four properties:

1. J is an invariant of the system: the concrete and the abstract data structures are always related by J.

   We satisfy this property by construction. When inferring J, we start with the inferred postcondition of the constructor `init()` and do a fixpoint calculation to compute a weaker assertion that is stable under interference from

the given actions. This fixpoint calculation is also known as *stabilization*. For more details about how stabilization is done, see [14]. As the actions soundly overapproximate the system, this implies that the inferred assertion is an invariant of the system.
2. In every trace representing the execution (whether terminating or not) of a method, there is *at most one* linearization point of that method call.
3. Every terminating execution trace of a method has *at least one* linearization point.
4. Whenever a method terminates, it returns the same result as the specification embedded at the linearization point.

Putting (2) and (3) together means that terminating executions must have exactly one linearization point. In cases where the abstract code specifying a method has no side-effects (e.g. when `pop` returns `EMPTY`), we can drop condition (2). Dropping (2) typically reduces the annotation overhead for read-only methods because we do not need to ensure that the abstract effect of the method was executed exactly once.

Checking conditions (2) and (3) may seem trivial for Treiber's stack, but can be quite difficult in general because the linearization point along some execution paths of a method may be within code performed by another concurrently executing thread. This case arises frequently in methods that have no side-effects, and in algorithms that use 'helping.'

We verify conditions (2), (3), and (4) with a simple intentional encoding. For each method call, we create an auxiliary descriptor record with one field containing the name of the method, one field for each argument of the method, and one additional field, `ABS_RESULT`, which is assigned at the linearization point. At the beginning of each method, we add auxiliary code that allocates a new such record in the heap and initializes its fields. To check that the linearization point happens at most once, we initialize `ABS_RESULT` with a certain reserved value `UNDEF`. At the linearization points we check that `ABS_RESULT` still contains this special value and update it with the result of the abstract operation. At the method's return point, we check that the value returned is the same as the one stored in `ABS_RESULT` (and different than `UNDEF`). This ensures that the linearization point occurred exactly once.

As the auxiliary record is stored in the heap, it can be shared, and hence, a different thread can execute the auxiliary code that updates the `ABS_RESULT` field. Thus, we are able to handle methods whose linearization points along some executions are in a different thread.

Our use of the reserved value `UNDEF` encodes whether the linearization point has occurred or not. Alternatively, the same information can be recorded by a boolean variable. Gao et al. [3] instead keep a counter initially set to 0, incremented at each linearization point, and prove that it contains 1 at the end of the method. Besides using more state than necessary, their approach does not imply property (2) for non-terminating executions.

4 Shape-Value Abstraction

Most shape analyses abstract away the values stored in the data structures. This renders them practically useless for proving linearizability because the crucial invariant needed in order to prove linearizability is that the concrete data structure represents the same value as the abstract state.

A possible solution to this problem is to develop a specialized abstract domain that can express this invariant. This approach was followed by Amit et al. [6], who presented an abstract domain tracking graph isomorphism. Here, we will consider a different, possibly more general, solution.

Our abstraction follows a two step approach. First we abstract the shapes of the data structures, and then we abstract the values stored in those data structures. These two steps are independent to each other, and hence we can combine any suitable shape abstraction with any suitable value abstraction. Formally, our abstraction function is the composition of two abstractions:

$$\alpha_{\mathsf{total}} = \alpha_{\mathsf{value}} \circ \alpha_{\mathsf{shape}}$$

The function α_{shape} handles shape-related issues, whereas the function α_{value} handles value-related issues. Correspondingly, the concretization function is the composition of the two corresponding concretization functions:

$$\gamma_{\mathsf{total}} = \gamma_{\mathsf{shape}} \circ \gamma_{\mathsf{value}}$$

This setup simplifies proving correctness of the analysis: we can prove separately that the two abstraction functions are correct.

In the following, the abstract domains are just subsets of the concrete domain; hence, the γ-functions are the corresponding inclusion (i.e. the identity) functions.

4.1 Shape Abstraction

Given a shape analysis based on separation logic, deriving the shape abstraction (α_{shape}) is straightforward. The shape analysis's abstraction function can be decomposed in two more primitive functions: one that abstracts shape information, but treats values precisely, and a second one that abstracts all value-related information.

We proceed with a concrete example. We derive a value-remembering shape abstraction from the shape analysis of Distefano et al. [15]. Distefano's analysis is based on separation logic, and handles singly linked data structures. Their abstract domain is a subset of separation logic assertions that includes $*$-conjunction, disjunction, $\mapsto$, emp, junk, lseg, equalities and disequalities. The assertion emp denotes the empty heap (in which nothing is allocated); junk is true for any heap, whether empty or consisting of some allocated nodes. Finally, the predicate $\mathsf{lseg}(x, y)$ denotes a singly linked list segment starting at address x and ending at y. For technical reasons (see [14] for details), we prefer a slightly different version of the list segment predicate, whose inductive definition is given below:

$$\mathsf{lseg}(x, y) \stackrel{\text{def}}{=} (x = y \wedge \mathsf{emp}) \vee (\exists bz.\ \mathsf{Node}(x, z, b) * \mathsf{lseg}(z, y))$$

where $\mathsf{Node}(x, y, v) \stackrel{\text{def}}{=} x \mapsto \{\texttt{.next} = y, \texttt{.data} = v\}$.

$$\begin{aligned}
\mathsf{Node}(y,z,b) &\implies \mathsf{junk}\\
\mathsf{Node}(x,y,a) * \mathsf{Node}(y,z,b) &\implies \mathsf{lseg}_{\mathsf{new}}(x,z,\langle a\rangle{\cdot}\langle b\rangle)\\
\mathsf{lseg}_{\mathsf{new}}(x,y,a) * \mathsf{Node}(y,z,b) &\implies \mathsf{lseg}_{\mathsf{new}}(x,z,a{\cdot}\langle b\rangle)\\
\mathsf{lseg}_{\mathsf{new}}(y,z,b) &\implies \mathsf{junk}\\
\mathsf{Node}(x,y,a) * \mathsf{lseg}_{\mathsf{new}}(y,z,b) &\implies \mathsf{lseg}_{\mathsf{new}}(x,z,\langle a\rangle{\cdot}b)\\
\mathsf{lseg}_{\mathsf{new}}(x,y,a) * \mathsf{lseg}_{\mathsf{new}}(y,z,b) &\implies \mathsf{lseg}_{\mathsf{new}}(x,z,a{\cdot}b)
\end{aligned}$$

Fig. 2. Shape abstraction rules

We can extend the list segment predicate with an additional argument recording the sequence of values represented by the list.

$$\begin{aligned}
\mathsf{lseg}_{\mathsf{new}}(x,y,a) \stackrel{\text{def}}{=}\ & (x = y \wedge a = \epsilon \wedge \mathsf{emp})\\
& \vee\ \exists bcz.\ a = \langle b\rangle{\cdot}c * \mathsf{Node}(x,z,b) * \mathsf{lseg}_{\mathsf{new}}(z,y,c)
\end{aligned}$$

Distefano's abstraction function consists of applying a set of rewrite rules as much as possible. Each rewrite rule is a valid separation logic implication, and eliminates one existentially quantified variable from the input assertion. This ensures that the abstraction function is sound and always terminates. Distefano also proves that his abstract domain is finite; hence, fixpoints in the abstract domain converge.

Our abstraction has the same structure, but we have modified the rewrite rules to record value-related information accurately (see Fig. 2). For example, our last rule is a direct adaptation of Distefano's rule for merging two list segments:

$$\mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) \implies \mathsf{lseg}(x,z).$$

Abstraction applies these rules aggressively whenever y is an existentially quantified variable that does not appear in the rest of the formula. Abstraction is sound, because each rewrite rule is a valid separation logic implication.

In essence, we have decomposed Distefano's abstraction function $\alpha_{\mathsf{Distefano}}$ into two steps, $\alpha_{\mathsf{Distefano}} = \alpha_{\mathsf{forget\_values}} \circ \alpha_{\mathsf{shape}}$, where $\alpha_{\mathsf{forget\_values}}$ maps every $\mathsf{lseg}_{\mathsf{new}}(x,y,v)$ into $\mathsf{lseg}(x,y)$. Shape-value abstraction will keep the α_{shape} part, but replace the $\alpha_{\mathsf{forget\_values}}$ function with something more appropriate.

4.2 Value Abstraction

Now we turn to the abstraction of values appearing in a formula. Recall that the basic invariant in a linearizability proof is that two data structures represent the same value. Therefore, we want an abstraction that remembers some correlations between equal values. To be concrete, consider we want to abstract the values in the following assertion:

$$\mathsf{lseg}(k,0,b{\cdot}c{\cdot}d{\cdot}e) * \mathsf{lseg}(l,0,a{\cdot}b) * \mathsf{lseg}(m,0,a{\cdot}b) * \mathsf{lseg}(n,0,e).$$

There are three natural choices as to what abstraction should do:

A. Keep track of the equalities between top-level expressions (such as $a{\cdot}b$):

$$\exists uvw.\ \mathsf{lseg}(k, 0, u) * \mathsf{lseg}(l, 0, v) * \mathsf{lseg}(m, 0, v) * \mathsf{lseg}(n, 0, w)$$

B. Also keep track of the correlations between a top-level expression and subexpressions of another expression (such as e).

$$\exists uvw.\ \mathsf{lseg}(k, 0, u{\cdot}w) * \mathsf{lseg}(l, 0, v) * \mathsf{lseg}(m, 0, v) * \mathsf{lseg}(n, 0, w)$$

C. Also keep track between any two subexpressions (such as b).

$$\exists tuvw.\ \mathsf{lseg}(k, 0, u{\cdot}v{\cdot}w) * \mathsf{lseg}(l, 0, t{\cdot}u) * \mathsf{lseg}(m, 0, t{\cdot}u) * \mathsf{lseg}(n, 0, w)$$

It turns out that choice A is too weak for linearizability proofs, and that we need one of the other two choices. In particular, choice A can prove the linearizability of `push`, but not of `pop`. In the proof outline of `pop`, one of the disjuncts of the assertion between lines `15` and `16` of `pop` is

$$\exists \alpha\beta.\ \mathtt{S} \mapsto \mathtt{Top:t} * \mathtt{t} \mapsto \mathtt{data:}\alpha\mathtt{,next:x} * \mathsf{lseg}(\mathtt{x}, 0, \beta) * \mathtt{ABS} \mapsto \mathtt{val:}\langle\alpha\rangle{\cdot}\beta$$

In this case, the first choice would forget the correlation between the value between the concrete data structure and `ABS->val`, which would make it impossible to prove that the concrete `pop` returns the same result as the abstract `pop`.

In our example programs, choice B was sufficient for proving linearizability. Choice C also works, but as it distinguishes more abstract states, it is potentially slower. A benefit of choice C is that it is more robust against more aggressive shape abstractions. Considering syntactic subexpressions is not sufficient, but one has to take the properties (such as associativity and commutativity) of the value constructors into account.

Our general approach for performing value abstraction works as follows. First, we collect the set T of all values appearing in the formula. From that set, we deduce a set of values, S, that we will 'forget' (i.e. existentially quantify over). For each value v_i in S, we introduce a fresh existentially quantified variable x_i, and we (back-)substitute x_i for v_i in the assertion. This abstraction is sound irrespective of S, because $P(v_1, \ldots, v_n) \implies \exists x_1, \ldots, x_n.\ P(x_1, \ldots, x_n)$.

The way we select S is crucial for the precision of the analysis. To get choice A, simply choose $S = T$. To get the other two choices, more work is necessary. Below, we consider this additional work for two kinds of values: (*i*) sets and multisets, and (*ii*) strings/sequences.

Sets & Multisets. Consider expressions denoting sets or multisets constructed using the operations: empty set/multiset, singleton set/multiset, and set/multiset union. (We shall ignore intersection and difference operators.) To take care of the associativity and commutativity of $\cup$, we represent set expressions canonically as a union of a set of expressions and we have a special constructor for singleton sets. For example, the set expression $\{1, 2\} \cup (x \cup y)$ would be represented as

$\{singleton(1), singleton(2), x, y\}$. Then, in order to get choice C, we compute the set S of set expressions according to the following algorithm:

$$
\begin{array}{l}
S := T \setminus \{\emptyset\}; \\
\textbf{while } \exists x, y \in S.\ x \neq y \ \wedge\ x \cap y \neq \emptyset \textbf{ do} \\
\quad S := (S \setminus \{x, y\}) \cup (\{x \setminus y, x \cap y, y \setminus x\} \setminus \{\emptyset\})
\end{array}
$$

We start with T, the set of all (set) values appearing in the formula. In the loop, while there exist overlapping sets in S, we remove them from S and add the three partitions. At the end, all the elements S will be disjoint, and any element of T denoting a set will be expressible as a union of elements in S. Notice that these rewrites are confluent: the choice of x and y at each loop iteration does not affect the final result.

Here is our algorithm as applied to a small example:

Initial configuration:	$\{1,2,3\}, \{1,4,5\}, \{2,3,6\}$.
Choosing $x = \{1,2,3\}$ and $y = \{1,4,5\}$ yields	$\{1\}, \{2,3\}, \{4,5\}, \{2,3,6\}$.
Choosing $x = \{2,3\}$ and $y = \{2,3,6\}$ yields	$\{1\}, \{2,3\}, \{4,5\}, \{6\}$.
No further loop iterations are possible.	

To get choice B, we also require that either $x \subseteq y$ or $y \subseteq x$.

Sequences. Sequences are strings over the alphabet of expressions. They are built out of three operations: the empty sequence (ϵ), the singleton sequence (which we write $\langle x \rangle$) and concatenation (denoted $x{\cdot}y$). Analogously to sets, the analysis represents sequence expressions as a sequence of expressions that are concatenated together. To get choice C, we compute S as follows:

$$
\begin{array}{l}
S := T \setminus \{\epsilon\}; \\
\textbf{while } \left(\begin{array}{l} \exists x \in S, y \in S.\ \exists z, x_1, x_2, y_1, y_2. \\ \qquad x \neq y \wedge z \neq \epsilon \wedge x = x_1{\cdot}z{\cdot}x_2 \wedge y = y_1{\cdot}z{\cdot}y_2 \end{array} \right) \textbf{ do} \\
\quad S := (S \setminus \{x, y\}) \cup (\{x_1, x_2, y_1, y_2, z\} \setminus \{\epsilon\})
\end{array}
$$

We start with T, the set of all values in the formula. In the loop, while there exists a non-empty common subsequence (z) in two elements of S, we remove those elements from S, and replace them with the partitions x_1, x_2, y_1, y_2, and z. To get choice B, we also require that either $x \sqsubseteq y$ or $y \sqsubseteq x$, where $x \sqsubseteq y$ holds if and only if there exist w_1 and w_2 such that $y = w_1{\cdot}x{\cdot}w_2$. Equivalently, to get choice B, we require that either $x_1 = x_2 = \epsilon$ or $y_1 = y_2 = \epsilon$.

Unlike the set/multiset algorithm, different instantiations of the existential variables can lead to different final results. This is problematic because some results are better than others (we want to minimize the cardinality of the final S so that we do not accidentally miss any abstraction opportunites). Fortunately, a simple condition ensures that the best result is found: the z selected must be a (local) maximum. Formally, for all z', if $z \sqsubseteq z'$, then $z' \not\sqsubseteq x$ or $z' \not\sqsubseteq y$. Ensuring this condition is an easy programming task.

5 Extensions to RGSep

We have implemented our abstraction function in a static analyzer based on RGSep [13,14]. In this section, we will briefly go over the key concepts of RGSep, and show how we modified its atomic rule to deal with instructions such as CAS.

In RGSep, the state of the program is logically divided into a static number of (disjoint) partitions, which are called regions. Each thread of the system owns one region for its local data, and there are also regions containing data that is shared among threads.

The program logic permits each thread to access local state directly, and restricts shared state accesses to use some form of synchronisation (e.g. mutexes, atomic reads, CAS). At synchronisation points, the thread can re-adjust the boundaries between local and shared state. Whenever a thread modifies the shared state (or the partitioning of the shared state), the logic ensures that the correctness of the other threads is resistant to the modification. This is achieved with rely/guarantee reasoning.

In particular, the concurrent behaviour of each thread is abstracted by a set of precondition-postcondition pairs, known as *actions*. These actions summarise what modifications the atomic statements of a thread can perfom on the shared state.

For each atomic statement of a thread, Calcagno et al. [14] check that there is an action abstracting its entire effect. This is sufficient if all the atomic blocks consist of a single memory access, but is awkward for larger atomic statements such as CAS. CAS has a conditional effect: if it reads the expected value, then it modifies the state; else it does nothing. We can write an action that captures this complex effect, but it will be quite complex itself. For instance, the `Apush` action from Sect. 2 would have to use a postcondition with a disjunction, encoding the two possibilities of the CAS:

$$\begin{array}{l} \texttt{\_x==0} * \texttt{S}{\mapsto}\texttt{Top:}y * y{\mapsto}\texttt{data:}e\texttt{,next:}n * \texttt{ABS}{\mapsto}\texttt{val:}\langle e\rangle{\cdot}A \\ \texttt{||}\ \texttt{\_x!=0} * \texttt{S}{\mapsto}\texttt{Top:}n * \texttt{ABS}{\mapsto}\texttt{val:}A \end{array}$$

Not only is the action unnecessarily long (and therefore difficult to specify or to infer), but it also slows down stabilization. Stabilization is an expensive computation that is executed after the symbolic execution of every atomic command. Given an assertion, it does a fixpoint calculation to compute a weaker assertion that it stable under the set of given actions. Its execution time is roughly proportional to the size of the action definitions.

Instead we allow actions to specify parts of an atomic statement. For example, the actions of Section 2 describe only the effects of successful CASes. We change the input language of Calcagno et al. [14] by dropping action annotations from atomic statements and adding a new form of statement for action annotation (the '`do...as`' block). We impose a syntactic restriction that these '`do...as`' blocks can appear only inside atomic blocks.

In the proof rules below, the judgement $\{P_0 \mid P_1\}\ C\ \{Q_0 \mid Q_1\}$ says that the program C has local precondition P_0, shared precondition P_1 local postcondition Q_0 and shared postcondition Q_1. (In reality, we have an indexed family of shared

preconditions and shared postconditions, but we will describe our rules as if there was only one for simplicity.) Symbolic execution takes P_0, P_1, and C as arguments and computes (strongest) Q_0 and Q_1. Normally, commands can access only the local state P_0. As an exception, memory reads *inside an atomic block* can also access the shared state:

$$\{P \mid \texttt{e} \mapsto \texttt{field}{:}e' * Q\}\ \texttt{x = e->field;}\ \{\texttt{x} = e' * P \mid \texttt{e} \mapsto \texttt{field}{:}e' * Q\}$$

Unlike Calcagno et al., our symbolic execution does nothing at entries to atomic blocks. At exits, it computes a weaker shared postcondition that is resistant to interference from other threads.

$$\frac{\{P_0 \mid P_1\}\ \texttt{C}\ \{Q_0 \mid Q_1\}}{\{P_0 \mid P_1\}\ \texttt{atomic C}\ \{Q_0 \mid \mathit{stabilize}(Q_1)\}}$$

When symbolic execution encounters an action annotation, it has more work to do. At the beginning of the block, it removes the precondition P of the action from the shared state, and adds it to the local state. Correspondingly, at the end of the block it removes the postcondition Q of the action from the local state and adds it to the shared state.

$$\frac{\{P_0 * P \mid P_2\}\ \texttt{C}\ \{Q_0 * Q \mid Q_2\}}{\{P_0 \mid P * P_2\}\ \texttt{do C as}_{P \rightsquigarrow Q}\ \{Q_0 \mid Q * Q_2\}}$$

This ensures that the annotated action accounts for any change that `C` makes to the shared state. Therefore, as the shared state can be changed only within `do...as` blocks, the set of annotated actions covers every possible shared state change that the program can make.

Experience suggests that writing these action annotations is straightforward and that the process of figuring out the correct actions has a very local, syntactic nature. It is possible that in many simple cases these annotations can be inferred automatically, but we have not investigated this possibility yet.

6 Evaluation

Table 1 presents our experimental results. We verify a number of concurrent algorithms from the literature.

The first four algorithms do not leak memory. The DCAS stack is similar to Treiber's stack (presented in Section 2), but `pop` uses a double compare-and-swap instruction instead of a single CAS. The two-slot buffer is an obstruction-free implementation of an atomic register with a single reader and a single writer. The two-lock queue is due to Michael and Scott [16] and uses two locks: one for protecting the head of the list, and one for the tail of the list. The lock-coupling list [5] represents a set of integers as a sorted linked list with one lock per node. When traversing the list, locks are acquired in a hand-over-hand fashion. The available operations are single element addition, removal, and test of membership. As we have not implemented an abstraction for sorted lists, we currently verify that when these operations succeed, they are the correct multiset operations.

Table 1. Verification times for a collection of concurrent algorithms

Data structure	**Shape analysis**	**Linearizability**	**Berdine et al.** [8]
DCAS stack	0.2s	0.2s	–
Two-slot buffer	0.4s	1.2s	–
Two-lock queue [16]	0.5s	0.6s	17s
Lock-coupling list [5]	0.3s	0.5s	–
Treiber stack [12]	0.2s	0.3s	7s
Non-blocking queue [16]	2.6s	5.1s	–
Non-blocking queue [2]	2.6s	4.8s	252s
RDCSS [17]	1.6s	87.7s	–

The next four algorithms have memory leaks and depend on a garbage collector for correctness. Treiber's stack was presented in Sect. 2. The first non-blocking queue algorithm is the well-known Michael and Scott's queue [16]. The second non-blocking queue algorithm is a slight variation which was verified by Doherty et al. [2]. Finally, RDCSS [17] is a lock-free implementation of restricted double-compare single-swap primitive. Proving linearizability of RDCSS is challenging because some of its linearization points are executed by different threads, and specifying them requires an auxiliary prophecy variable.

Each column records verification time in seconds. Our tests were conducted on a 3.4GHz Pentium 4 processor running Windows Vista. In all cases, memory consumption was under 5 megabytes. The first column measures the time required by the underlying shape analysis. This infers the shape of the data structures used in the heap and checks that there are no memory errors (e.g. null pointer dereferences). For the first four algorithms, it also checks that there are also no memory leaks. The second column measures the total time required to prove linearizability using the techniques described in this paper. The difference between these two columns represents the additional amount of work that is needed in order to prove linearizability.

Finally, the last column displays the results of Berdine et al. [8]. Comparison with this work is purely indicative; direct comparison is unfair because the tools are quite different. We used the same shape abstraction for all the examples, but require actions to be annotated. In contrast, Berdine et al. do not require any action annotations, but use slightly different abstractions for each algorithm and require an user-supplied heap decomposition.

We have also performed tests where we inserted errors in the algorithms. In all these cases, our tool failed to prove linearizability.

7 Related Work

Automatic Verification. Wang and Stoller [18] present a static analysis that verifies linearizability for an unbounded number of threads. Their analysis essentially detects certain coding patterns, which are known to be atomic irrespective of

the environment. Algorithms such as Michael and Scott's non-blocking queue that do not follow these coding patterns have to be rewritten.

Amit et al. [6] presented a shape difference abstraction that tracks the difference between two heaps. This approach works well if the concrete heap and the abstract heap have almost identical shapes during the entire algorithm. If, however, we are verifying a concurrent tree algorithm that rebalances the tree every so often, then the concrete heap and the abstract heap may differ dramatically regarding their shape, but not the values stored. In such cases, any abstraction requiring that the two heaps are isomorphic will fail completely. More recently, Manevich et al. [7] and Berdine et al. [8] have presented some improvements to this analysis, which are orthogonal to the task of verifying linearizability.

Finally, Yahav and Sagiv [19] and Calcagno et al. [14] use shape analysis to check simple safety properties of list-based concurrent algorithms, but cannot verify linearizability.

Semi-automatic Verification. In [2,3,4], the PVS theorem prover was used to check hand-crafted linearizability proofs. These papers prove linearizability using different techniques than the one used here. See Sect. 3 for details.

8 Conclusion

We have demonstrated that RGSep and shape-value abstraction enable effective automatic linearizability proofs. The examples verified are typical of the research literature 5–10 years ago. The techniques can also cope with more complex algorithms, but the shape analyses must be powerful enough to describe the data structures used in the algorithms. As shape analyses based on separation logic are relatively new, they are still restricted to linked-list data structures.

We believe that both further instances of shape-value abstraction as well as the presented value abstractions apply equally to other verification problems, but we have not investigated this possibility yet.

In the future, we plan to improve the underlying shape analyses to handle other kinds of data structures such as arrays, and to attempt to infer the necessary action annotations automatically.

Acknowledgments. We would like to thank Alan Mycroft, Hongseok Yang and the anonymous referees for providing helpful feedback on earlier drafts of this work.

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. on Program. Languages and Systems 12(3), 463–492 (1990)
2. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)

3. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. Distributed Computing 18(1), 21–42 (2005)
4. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 473–488. Springer, Heidelberg (2006)
5. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP 2006, pp. 129–136. ACM Press, New York (2006)
6. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
7. Manevich, R., Lev-Ami, T., Ramalingam, G., Sagiv, M., Berdine, J.: Heap decomposition for concurrent shape analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 363–377. Springer, Heidelberg (2008)
8. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
9. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
10. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
12. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr. (1986)
13. Vafeiadis, V.: Fine-grained concurrency verification. Ph.D. dissertation, University of Cambridge (2007); Also available as Technical Report UCAM-CL-TR-726
14. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007)
15. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
16. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, pp. 267–275. ACM Press, New York (1996)
17. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 265–279. Springer, Heidelberg (2002)
18. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: PPoPP 2005, pp. 61–71. ACM Press, New York (2005)
19. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. Electronic Notes in Theoretical Computer Science 89(3) (2003)

Mixed Transition Systems Revisited

Ou Wei[1], Arie Gurfinkel[2], and Marsha Chechik[1]

[1] Department of Computer Science, University of Toronto
[2] Software Engineering Institute, Carnegie Mellon University

Abstract. Partial models support abstract model-checking of complex temporal properties by combining both over- and under-approximating abstractions into a single model. Over the years, three families of such modeling formalisms have emerged, represented by Kripke Modal Transition Systems (KMTSs), with restrictions on necessary and possible behaviors, Mixed Transition Systems (MixTSs), with relaxation on these restrictions, and Generalized Kripke MTSs (GKMTSs), with hyper-transitions, respectively. In this paper, we compare the three families w.r.t. their expressive power (i.e., what can be modeled, what abstraction can be captured), and the cost and precision of model-checking. We show that these families have the same expressive power (but do differ in succinctness), whereas GKMTSs are more precise (i.e, can establish more properties) for model-checking than the other two families. However, the use of GKMTSs in practice has been hampered by the difficulty of encoding them symbolically. We address this problem by developing a new semantics for temporal logic of partial models that makes the MixTS family as precise for model-checking as the GKMTS family. The outcome is a symbolic model-checking algorithm that combines the efficient symbolic encoding of MixTSs with the model-checking precision of GKMTSs. Our preliminary experiments indicate that the new algorithm is a good match for predicate-abstraction-based model-checkers.

1 Introduction

Abstraction is the key to scaling model-checking to industrial-sized problems. Typically, a large (or infinite) concrete system is approximated by a smaller abstract system via abstracting the concrete states, analyzing the resulting abstract system, and lifting the result back to the concrete system. Two common abstraction schemes are *over-approximation* – the abstract system contains *more* behaviours than the concrete one and is sound for universal properties (e.g., absence of errors), and *under-approximation* – the abstract system contains *less* behaviours than the concrete one and is sound for existential properties (e.g., presence of errors). Abstractions that are sound for arbitrary properties such as full μ-calculus L_μ [14], must combine over- and under-approximation into a *single* model [4, 15]. This can be done by using a model with two types of transitions, *may* and *must*, representing *possible* (or over-approximating), and *necessary* (or under-approximating) behaviours, respectively. We call such models *partial*. Temporal properties over partial models are interpreted using the 3-valued semantics: a property can be either true, false, or *unknown*.

Existing partial modeling formalisms are divided into three separate families. The first is *Kripke Modal Transition Systems* (KMTSs) [13] and their equivalent variants,

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 349–365, 2009.

Modal TSs [15], *Partial Kripke Structures* (PKSs) [1], and *3-valued KSs* [2]. KMTSs require that every *must* transition is also a *may* transition. They were introduced as computational models for partial specifications of reactive systems [15] and then adapted for model-checking [1, 2, 13]. The second is *Mixed Transition Systems* (MixTSs) [4], and equivalently, *Belnap TSs* [11]. MixTSs extend KMTSs by allowing *must only* transitions (i.e., transitions that are *must* but not *may*). MixTSs were introduced in [4] as abstract models for L_μ, and have been used for predicate abstraction and software model-checking in [10]. The third is *Generalized KMTSs* (GKMTSs) [19], and equivalently, *Abstract TSs* [6] and *Disjunctive MTSs* [16]. GKMTSs extend MixTSs by allowing *must hyper-transitions*, (i.e., transitions into sets of states).

In this paper, we compare the three families w.r.t. their suitability as the "right" formalism for symbolic model-checking of partial models. Our basis of comparison is *(i)* the expressive power of the formalisms (i.e., what can be modeled, what abstraction can be captured), and *(ii)* analyzability of the formalisms (i.e., the cost and precision of model-checking).

Expressive Power. We show that MixTSs, KMTSs and GKMTSs are equally expressive: for any partial model M expressed in one formalism, there exists a partial model M' in the other s.t. M and M' approximate the same set of concrete systems. That is, neither hyper-transitions nor restrictions on *may* and *must* transitions affect expressiveness. They do, however, affect the size of the models: GKMTSs and KMTSs can be converted to semantically equivalent MixTSs of (possibly exponentially) smaller or equal size. Dams and Namjoshi have shown that all of the above partial models are less expressive than tree automata [5]. We complete the picture by showing the expressive equivalence *between* those formalisms.

Model Checking. We call a semantics of temporal logic *inductive* if it is defined inductively on the syntax of the logic. We refer to the typical inductive semantics of L_μ on partial models as *standard* inductive semantics (SIS). This is the semantics most widely used in practice. A GKMTS G can prove/disprove more properties under SIS than either a corresponding MixTS M or KMTS K obtained from G by semantics-preserving translations. However, while both MixTSs and KMTSs have been used in practical symbolic model-checkers (e.g., [2, 10, 12]), the direct use of GKMTSs has been hampered by the difficulty of encoding hyper-transitions into BDDs. To address this problem, we develop a new semantics, called *reduced* (RIS), that is inductive (and tractable) but is more precise than SIS. We show that GKMTSs and MixTSs are equivalent w.r.t. RIS, and give an efficient symbolic model-checking procedure for RIS. The outcome is an algorithm that combines the benefits of the efficient symbolic encoding of MixTSs with the model-checking precision of GKMTSs.

To show the practicality of the above result, we develop a symbolic model-checking algorithm w.r.t. to RIS and apply it to MixTS models constructed using predicate abstraction. We evaluate our implementation empirically against a SIS-based algorithm.

The rest of the paper is organized as follows. Sec. 2 reviews the necessary background on partial models and abstraction. In Sec. 3, we prove that KMTSs, MixTSs and GKMTSs are equally expressive by developing semantics-preserving translations from GKMTSs to MixTSs, and from MixTSs to KMTSs. In Sec. 4, we introduce *reduced*

inductive semantics (RIS) for L_μ. In Sec. 5, we present a symbolic model-checking algorithm w.r.t. RIS in the context of predicate abstraction. We report on our experience with this algorithm in Sec. 6. Sec. 7 concludes the paper with a summary and comparison with related work.

2 Preliminaries

In this section, we review several modeling formalisms, and their use for abstraction.

2.1 Complete and Partial Models

A statespace of a *partial* transition system is a tuple $\langle S, \preceq_S \rangle$, where S is a set of states, and $\preceq_S$ is a partial order on S. Intuitively, $s_1 \preceq_S s_2$ means that s_1 is less informative (more partial) than s_2. For brevity of notation, we denote a statespace using the set S.

Definition 1 (Partial TSs). *[4, 8, 13, 19] A* Generalized Kripke Modal Transition System *(GKMTS) is a tuple $M = \langle S, R^{\text{may}}, R^{\text{must}} \rangle$, where S is the statespace, and $R^{\text{may}} \subseteq S \times S$, $R^{\text{must}} \subseteq S \times 2^S$ are the* may *and* must *transition relations, respectively. A* Mixed TS *(MixTS) is a GKMTS s.t. $R^{\text{must}} \subseteq S \times S$. A* Kripke Modal TS *(KMTS) is a MixTS s.t. $R^{\text{must}} \subseteq R^{\text{may}}$. A* Boolean TS *(BTS) is a KMTS s.t. $R^{\text{may}} = R^{\text{must}}$.*

We write $s \xrightarrow{\text{may}} t$ for $(s,t) \in R^{\text{may}}$, $s \xrightarrow{\text{must}} t$, and $s \xrightarrow{\text{must}} Q$ for $(s,t) \in R^{\text{must}}$ and $(s,Q) \in R^{\text{must}}$, respectively. Intuitively, *may* and *must* transitions represent possible and necessary behaviours, respectively. For example, a BTS is *complete* (i.e., not partial) since every *may* behaviour is also a *must* behaviour.

Let AP be a set of atomic propositions, $Lit(AP)$ be a set of literals of AP, and S be a statespace. A *state labeling* is a function $L : S \to 2^{Lit(AP)}$ that assigns to each state s a set of literals that are true in s. For a proposition p, if $p \in L(s)$, we say that p is true in s; if $\neg p \in L(s)$ — p is false in s; otherwise, the value of p is *unknown*. We require that a state labeling is *locally consistent*, i.e., at most one of p and $\neg p$ belongs to $L(s)$; and *monotone* w.r.t. $\preceq_S$, i.e., $s_1 \preceq_S s_2 \Rightarrow L(s_1) \subseteq L(s_2)$. A pair $\langle M, L \rangle$ of a TS M and a labeling L is called a *model*.

The modal μ-calculus [14] (L_μ) is defined as the set of all formulas satisfying the following grammar: $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Diamond\varphi \mid \mu Z \cdot \varphi(Z)$, where p is an atomic proposition, and Z a fixpoint variable. Furthermore, Z in $\mu Z \cdot \varphi(Z)$ must occur under the scope of an even number of negations. Additional operations are defined as abbreviations: $\varphi \vee \psi \triangleq \neg(\neg\varphi \wedge \neg\psi)$, $\Box\varphi \triangleq \neg\Diamond\neg\varphi$, $\nu Z \cdot \varphi(Z) \triangleq \neg\mu Z \cdot \neg\varphi(\neg Z)$. Let $\mathcal{M} = \langle M, L \rangle$ be a model, where $M = \langle S, R^{\text{may}}, R^{\text{must}} \rangle$, and φ be an L_μ formula. An *interpretation* (or *semantics*) of φ over $\mathcal{M}$, denoted $\|\varphi\|^{\mathcal{M}}$, is given by a pair $\langle U, O \rangle$, where $U, O \subseteq S$. Intuitively, U is the set of states that satisfy φ, and O is the set of states that do not refute φ. Thus, φ is true in U, false in $S \setminus O$ and unknown in $O \setminus U$. We call U and O the *under-* and the *over-approximation* of φ, respectively.

A semantics of L_μ is called *inductive* if it is inductive on the syntax of the logic. We refer to the commonly used inductive semantics as *standard* (SIS). We need the following notation. Let $e = \langle U, O \rangle$. We write $\mathsf{U}(e)$ and $\mathsf{O}(e)$ to denote U and O, respectively; we use operators $\sim$ and $\sqcap$ defined as follows: $\sim\langle U, O \rangle \triangleq \langle \overline{O}, \overline{U} \rangle$, and $\langle U_1, O_1 \rangle \sqcap \langle U_2, O_2 \rangle \triangleq \langle U_1 \cap U_2, O_1 \cap O_2 \rangle$.

Definition 2 (SIS). *[4, 8, 13, 19, 11] Let $\mathcal{M} = \langle M, L_M \rangle$ be a model, $M = \langle S, R^{\text{may}}, R^{\text{must}} \rangle$, Var a set of fixpoint variables, and $\sigma : Var \to 2^S \times 2^S$. The* standard inductive semantics (SIS) *of $\varphi \in L_\mu$ is:*

$$
\begin{aligned}
||p||^{\mathcal{M}}_{c,\sigma} &\triangleq \langle \{s \mid p \in L_M(s)\}, \{s \mid \neg p \notin L_M(s)\} \rangle \\
||\neg\varphi||^{\mathcal{M}}_{c,\sigma} &\triangleq \sim||\varphi||^{\mathcal{M}}_{c,\sigma} \qquad ||\varphi \wedge \psi||^{\mathcal{M}}_{c,\sigma} \triangleq ||\varphi||^{\mathcal{M}}_{c,\sigma} \sqcap ||\psi||^{\mathcal{M}}_{c,\sigma} \qquad ||Z||^{\mathcal{M}}_{c,\sigma} \triangleq \sigma(Z) \\
||\Diamond\varphi||^{\mathcal{M}}_{c,\sigma} &\triangleq \langle pre_{\mathsf{U}}(\mathsf{U}(||\varphi||^{\mathcal{M}}_{c,\sigma})), pre_{\mathsf{O}}(\mathsf{O}(||\varphi||^{\mathcal{M}}_{c,\sigma})) \rangle \\
||\mu Z \cdot \varphi||^{\mathcal{M}}_{c,\sigma} &\triangleq \langle lfp^{\subseteq}\left(\lambda Q \cdot \mathsf{U}(||\varphi||^{\mathcal{M}}_{c,\sigma[Z \mapsto Q]})\right), lfp^{\subseteq}\left(\lambda Q \cdot \mathsf{O}(||\varphi||^{\mathcal{M}}_{c,\sigma[Z \mapsto Q]})\right) \rangle
\end{aligned}
$$

where $Z \in Var$, lfp is the least fixpoint, and the pre-image *operators pre_{U} and pre_{O} are defined as follows:*

$$
pre_{\mathsf{U}}(Q) \triangleq \begin{cases} \{s \mid \exists t \in Q \cdot s \xrightarrow{\text{must}} t\} & \text{if } M \text{ is a MixTS} \\ \{s \mid \exists U \subseteq Q \cdot s \xrightarrow{\text{must}} U\} & \text{if } M \text{ is a GKMTS} \end{cases}
$$

$$
pre_{\mathsf{O}}(Q) \triangleq \{s \mid \exists t \in Q \cdot s \xrightarrow{\text{may}} t\}
$$

2.2 Partial Models and Abstraction

Abstract Statespace. A *concrete* statespace C is a set of states s.t. for any $c \in C$ and state labeling L, $p \in L(c) \Leftrightarrow \neg p \notin L(c)$. An *abstract* statespace approximating C is a set of states S together with a *soundness* relation $\rho : C \times S$, where $(c, s) \in \rho$ means that s *ρ-approximates* c. ρ induces a *concretization* function $\gamma(s) \triangleq \{c \mid (c, s) \in \rho\}$, and an *approximation* ordering $\preceq_a \subseteq S \times S$ defined as $s \preceq_a t \Leftrightarrow \gamma(s) \supseteq \gamma(t)$. That is, $\gamma(s)$ is the set of all concrete states approximated by s, and $s \preceq_a t$ if s is *less precise* (more approximate) than t. For a set $Q \subseteq S$, we define $\gamma(Q) \triangleq \cup_{s \in Q} \gamma(s)$. Following [3], we require that $\preceq_a$ be a partial order (i.e., the order $\preceq_S$), and that S satisfy "the existence of a best approximation": $\forall c \in C \cdot \exists s \in S \cdot (\rho(c, s) \wedge \forall s' \in S \cdot \rho(c, s') \Rightarrow \gamma(s') \supseteq \gamma(s))$. We use an *abstraction* function $\alpha : C \to S$ to map each concrete element to its best approximation. The image of α is denoted by $\alpha[S] \triangleq \{\alpha(c) \mid c \in C\}$.

Predicate Abstraction. Let n be a natural number, and $P = \{p_1, \ldots, p_n\}$ be a set of quantifier-free first-order boolean predicates. A *monomial* is a conjunction of literals of P; a *minterm* is a monomial in which each variable p_i appears exactly once (either positively or negatively). We write $\text{Mon}(P)$ and $\text{MT}(P)$ for the set of all monomials and minterms of P, respectively. The set $\text{Mon}(P)$ is the domain of predicate abstraction. The soundness relation ρ_P is defined s.t. $(c, s) \in \rho_P$ iff $c \models s$, i.e., c satisfies all predicates in s; the abstraction $\alpha_P(c) \triangleq (\bigwedge_{c \models p_i} p_i) \wedge (\bigwedge_{c \not\models p_i} \neg p_i)$; $\alpha_P[\text{Mon}(P)] = \text{MT}(P)$; and the approximation ordering is reverse implication, $s \preceq_a t$ iff $s \Leftarrow t$.

Simulation. An approximation relation is extended from a statespace to transition systems using the concept of *mixed simulation*.

Definition 3 (Mixed Simulation). *[4] Let $M_1 = \langle S_1, R_1^{may}, R_1^{must} \rangle$ and $M_2 = \langle S_2, R_2^{may}, R_2^{must} \rangle$ be two MixTSs. $H \subseteq S_1 \times S_2$ is a* mixed simulation *between M_1 and M_2 if for any $(s_1, s_2) \in H$, the following two conditions hold:*

$$\forall t_1 \in S_1 \cdot s_1 \xrightarrow{\text{may}} t_1 \Rightarrow \exists t_2 \in S_2 \cdot s_2 \xrightarrow{\text{may}} t_2 \wedge (t_1, t_2) \in H$$
$$\forall t_2 \in S_2 \cdot s_2 \xrightarrow{\text{must}} t_2 \Rightarrow \exists t_1 \in S_1 \cdot s_1 \xrightarrow{\text{must}} t_1 \wedge (t_1, t_2) \in H$$

In this case, we say M_2 H-simulates M_1, *written* $M_2 \preceq_H M_1$.

Intuitively, M_2 simulates M_1 whenever M_2 is less precise about its behaviour than M_1. This definition generalizes to GKMTSs (c.f., [19]).

Let C and S be a concrete and abstract statespaces, respectively, and $\rho \subseteq C \times S$ be the soundness relation. A partial TS M over S *approximates* a BTS B over C (or, equivalently B *refines* M) iff M ρ-simulates B, $M \preceq_\rho B$. Let L_M and L_B be state-labellings for S and C, respectively. L_M *approximates* L_B, denoted $L_M \preceq_\rho L_B$, iff $\rho(c, s) \Rightarrow L_M(s) \subseteq L_B(c)$. A partial model $\mathcal{M} = \langle M, L_M \rangle$ *approximates* a concrete model $\mathcal{B} = \langle B, L_B \rangle$ (or, equivalently, $\mathcal{B}$ *refines* $\mathcal{M}$) iff $M \preceq_\rho B$, and $L_M \preceq_\rho L_B$.

Theorem 1. *[4] Let* $\mathcal{B} = \langle B, L_B \rangle$ *be a concrete model that refines a partial model* $\mathcal{M} = \langle M, L_M \rangle$, *and* $\varphi \in L_\mu$. *Then,* $\gamma(\mathsf{U}(\|\varphi\|_c^{\mathcal{M}})) \subseteq \mathsf{U}(\|\varphi\|_c^{\mathcal{B}})$, *and* $\mathsf{O}(\|\varphi\|_c^{\mathcal{B}}) \subseteq \gamma(\mathsf{O}(\|\varphi\|_c^{\mathcal{M}}))$.

That is, if φ is true (false) at a state a of $\mathcal{M}$, then it is true (false) at all states $\gamma(a)$ of $\mathcal{B}$.

Let $\mathbb{C}[\mathcal{M}]$ be the set of all concrete refinements of $\mathcal{M}$. Intuitively, $\mathbb{C}[\mathcal{M}]$ is the semantic meaning of $\mathcal{M}$. An interpretation of L_μ based on the semantic meaning of a partial model was introduced in [1] as *thorough semantics*. It is defined as follows: $\|\varphi\|_t^{\mathcal{M}} = \langle U, O \rangle$ iff $a \in U \Leftrightarrow \forall \mathcal{B} \in \mathbb{C}[\mathcal{M}] \cdot \gamma(a) \subseteq \mathsf{U}(\|\varphi\|_c^{\mathcal{B}})$, and $a \notin O \Leftrightarrow \forall \mathcal{B} \in \mathbb{C}[\mathcal{M}] \cdot \gamma(a) \subseteq \mathsf{U}(\|\neg\varphi\|_c^{\mathcal{B}})$.

To compare different interpretations of L_μ, we introduce two ordering relations on $2^S \times 2^S$. Let $e_1 = \langle U_1, O_1 \rangle$ and $e_2 = \langle U_2, O_2 \rangle$. We say that e_1 is *less informative* than e_2, written $e_1 \preceq_i e_2$ iff $U_1 \subseteq U_2$ and $O_2 \subseteq O_1$. We say that e_1 is *semantically less precise* than e_2, written $e_1 \preceq_a e_2$, iff $\gamma(U_1) \subseteq \gamma(U_2)$ and $\gamma(\overline{O_1}) \subseteq \gamma(\overline{O_2})$.

3 Expressiveness

We show that GKMTSs, MixTSs, and KMTSs are expressively equivalent. Two partial TSs M and M' are *semantically equivalent*, $M \equiv_a M'$, iff they have the same set of concrete refinements. Two modeling formalisms are *expressively equivalent* iff for every TS M from one formalism, there exists a TS M' from the other, s.t. $M \equiv_a M'$. The equivalence of the three formalisms is proved by defining semantics-preserving translations from GKMTSs to MixTSs, and from MixTSs to KMTSs. Since GKMTSs syntactically subsume KMTSs, the translation from KMTSs to GKMTSs is basically an identity map.

3.1 GTOM: Translation from GKMTSs to MixTSs

We present the translation GTOM that converts a GKMTS into a semantically equivalent MixTS. First, we illustrate the translation on a GKMTS G_1 in Fig. 1. G_1 is not a MixTS because of *must* hyper-transition $a_1 \xrightarrow{\text{must}} \{a_2, a_3\}$. This transition ensures that in every concrete BTS refining G_1, all states in $\gamma(a_1)$, i.e., those satisfying

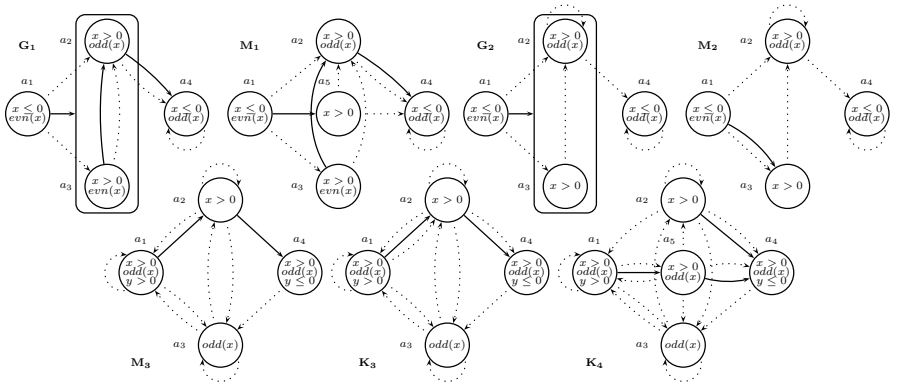


Fig. 1. Two GKMTSs: G_1, G_2; three MixTSs: M_1, M_2, M_3; two KMTSs: K_3, K_4. Solid and dashed lines represent must and may transitions, respectively.

$(x \leq 0 \wedge even(x))$, must have a transition to a state in $\gamma(\{a_2, a_3\})$, i.e., satisfying $(x > 0)$. No single state of G_1 represents $(x > 0)$. Thus, this requirement can only be captured either by a hyper transition (as done in G_1), or by extending G_1 with a new state, say a_5, such that $\gamma(a_5) = (x > 0)$. In the latter case, the *must* hyper-transition $a_1 \xrightarrow{\text{must}} \{a_2, a_3\}$ can be replaced by (regular) *must* transition $a_1 \xrightarrow{\text{must}} a_5$. The result is a MixTS M_1 in Fig. 1. Since a_5 replaces a "hyper-state" $\{a_2, a_3\}$, a_5 needs to preserve its *may* behaviours. This is done by adding $a_5 \xrightarrow{\text{may}} a_4$ and $a_5 \xrightarrow{\text{may}} a_2$ corresponding to $a_2 \xrightarrow{\text{may}} a_4$ and $a_3 \xrightarrow{\text{may}} a_2$, respectively. There are no outgoing *must* transitions from a_5 since the existing *must* transitions from a_2 and a_3 are sufficient. G_1 and M_1 are semantically equivalent: any BTS that refines G_1 also refines M_1, and vice versa.

In our example, a new state was added to encode a hyper-transition by a regular one. This isn't always necessary. For example, TSs G_2 and M_2 in Fig. 1 are semantically equivalent. The hyper-transition $a_1 \xrightarrow{\text{must}} \{a_2, a_3\}$ is encoded by $a_1 \xrightarrow{\text{must}} a_3$ in M_2 since the hyper-state $\{a_2, a_3\}$ is equivalent to an existing state a_3, i.e., $\gamma(\{a_2, a_3\}) = \gamma(a_3) = (x > 0)$.

In summary, a GKMTS G is translated to a MixTS M in two steps: (i) every *must* hyper-transition $a \xrightarrow{\text{must}} U$ of G is replaced by a regular *must* transition $a \xrightarrow{\text{must}} b$, where b is a (possibly new) state s.t. $\gamma(b) = \gamma(U)$; (ii) *may* transitions are added for every state introduced in the first step, if any. We formalize this below.

Definition 4 (GTOM). *Let $G = \langle S_G, R_G^{\text{may}}, R_G^{\text{must}} \rangle$ be a GKMTS. The translation* GTOM(G) *is a MixTS $M = \langle S_M, R_M^{\text{must}}, R_M^{\text{may}} \rangle$, such that*

$$S_M \triangleq S_G \cup S^+ \quad S^+ \triangleq \{a \mid \exists (s, U) \in R_G^{\text{must}} \cdot \gamma(a) = \gamma(U) \wedge (\forall t \in S_G \cdot \gamma(t) \neq \gamma(U))\}$$

$$R_M^{\text{may}} \triangleq R_G^{\text{may}} \cup \{(a, b) \mid a \in S^+ \wedge b \in S_G \wedge \exists s \in S_G \cdot (s, b) \in R_G^{\text{may}} \wedge \gamma(s) \subseteq \gamma(a)\}$$

$$R_M^{\text{must}} \triangleq \{(a, b) \mid a \in S_G \ \wedge \ b \in S_M \wedge \exists U \subseteq S_G \cdot (a, U) \in R_G^{\text{must}} \ \wedge \ \gamma(b) = \gamma(U)\}$$

The translation GTOM is semantics-preserving.

Theorem 2. *Let G be a GKMTS, and $M =$* GTOM(G)*. Then, M is a MixTS, and G and M are semantically equivalent.*

A corollary of Theorem 2 is that GKMTSs and MixTSs are equivalent w.r.t. thorough semantics. Let L_G be a labeling function for G. We extend the translation GTOM to a GKMTS model $\langle G, L_G\rangle$ such that $\text{GTOM}(\langle G, L_G\rangle) \triangleq \langle M, L_M\rangle$, where $M = \text{GTOM}(G)$, and L_M is a labeling function for S_M defined as follows:

$$L_M(a) \triangleq \begin{cases} L_G(a) & \text{if } a \in S_G \\ \bigcap_{\{s \in S_G \mid \gamma(s) \subseteq \gamma(a)\}} L_G(s) & \text{if } a \in S^+ \end{cases}$$

Then, L_M is well-defined and approximates the same labellings as L_G. This ensures that $\langle G, L_G\rangle$ and $\langle M, L_M\rangle$ satisfy the same properties under thorough semantics.

Corollary 1. *Let $\langle G, L_G\rangle$ be a GKMTS model and $\langle M, L_M\rangle = \text{GTOM}(\langle G, L_G\rangle)$. Then, $\langle G, L_G\rangle$ and $\langle M, L_M\rangle$ are equivalent w.r.t. thorough semantics.*

Complexity. We show that the translation GTOM does not increase the size of the model. Let G be a GKMTS with the statespace S_G, and $M = \text{GTOM}(G)$. The size of G is at most $|S_G \times 2^{S_G}|$. Each new state added by GTOM corresponds to a subset of S_G, i.e., $|S^+| \leq |2^{S_G}|$. Furthermore, no transitions between the states in S^+ are added. Thus, the size of M is also at most $|S_G \times 2^{S_G}|$.

Sometimes GTOM can reduce a GKMTS exponentially. For example, assume that S_G is a disjunctive completion [3], i.e., for every subset U of S_G there exists an equivalent element s in S_G such that $\gamma(U) = \gamma(s)$. In this case, GTOM does not add any new states, i.e., $S^+ = \emptyset$. This makes the size of the output MixTSs be $|S_G \times S_G|$, which is exponentially smaller than that of the input GKMTS.

3.2 MTOK: Translation from MixTSs to KMTSs

We present the translation MTOK that converts a MixTS into a semantically equivalent KMTS. First, we illustrate the translation using a MixTS M_3 in Fig. 1. M_3 is not a KMTS because of the two *must only* transitions $a_1 \xrightarrow{\text{must}} a_2$ and $a_2 \xrightarrow{\text{must}} a_4$. One way to turn M_3 into a KMTS is to add *may* transitions $a_1 \xrightarrow{\text{may}} a_2$ and $a_2 \xrightarrow{\text{may}} a_4$, resulting in K_3 in Fig. 1. This naive transformation is not semantics-preserving, i.e., $K_3 \not\equiv_a M_3$. For example, the concrete system[1]

$$\begin{aligned} &((y > 0) \wedge (x > 0) \wedge \mathit{odd}(x) \wedge x' = x + 1 \wedge y' = y) \vee \\ &\quad ((x > 0) \wedge \mathit{odd}(x) \wedge x' = x \wedge y' = -1 \times x) \vee \\ &\quad ((x > 0) \wedge \neg \mathit{odd}(x) \wedge x' = x + 1 \wedge y' = -1 \times x) \end{aligned}$$

refines K_3, but not M_3: the transition $\langle x = 1, y = 1\rangle \rightarrow \langle x = 2, y = 1\rangle$ cannot be simulated by any *may* transition of M_3.

The *must only* transition $a_1 \xrightarrow{\text{must}} a_2$ of M_3 ensures that in any concrete BTS refining M_3, all states in $\gamma(a_1)$, i.e., those satisfying $(x > 0 \wedge \mathit{odd}(x) \wedge y > 0)$, must have a transition to a state in $\gamma(a_2)$, i.e., satisfying $(x > 0)$. This is further restricted by the *may* transitions from a_1 that ensure that states in $\gamma(a_1)$ have transitions only to states in $\gamma(\{a_1, a_3\})$. Hence, in any BTS refining M_3, every state in $\gamma(a_1)$ must (and may) have a transition to a state in $\gamma(a_2) \cap \gamma(\{a_1, a_3\})$. That is, the restrictions posed by a *must only*

[1] Unprimed and primed variables represent current- and next-state valuations, respectively.

transition from a_1 are further restricted by the set of all of the *may* transitions from a_1. In general, for abstract states $b_0, \ldots, b_k$, a *must only* transition $b_0 \xrightarrow{\text{must}} b_1$, and a set of *may* transitions $b_0 \xrightarrow{\text{may}} b_2,\ \ldots,\ b_0 \xrightarrow{\text{may}} b_k$ ensure that every state in $\gamma(b_0)$ has a transition to a state in $\gamma(b_1) \cap \gamma(\{b_2, \ldots, b_k\})$.

The *must only* transition $a_2 \xrightarrow{\text{must}} a_4$ in M_3 is equivalent to a pair of *may* and *must* transitions from a_2 to a_4, since $\gamma(a_4) \cap \gamma(\{a_1, a_2, a_3\}) = \gamma(a_4)$. The *must only* transition $a_1 \xrightarrow{\text{must}} a_2$ can be equivalently represented by (a) adding a new state a_5 such that $\gamma(a_5) = \gamma(a_2) \cap \gamma(\{a_1, a_3\}) = (x > 0 \wedge odd(x))$, and (b) adding a *must* and a *may* transition from a_1 to a_5. Moreover, since a_5 approximates some of the same states as a_2, i.e., $\gamma(a_5) \subseteq \gamma(a_2)$, a_5 inherits the transitions from a_2: $a_5 \xrightarrow{\text{may}} a_1$, $a_5 \xrightarrow{\text{may}} a_2$, $a_5 \xrightarrow{\text{may}} a_3$, $a_5 \xrightarrow{\text{must}} a_4$, $a_5 \xrightarrow{\text{may}} a_4$. The final result is the KMTS K_4 in Fig. 1, which is semantically equivalent to M_3.

In summary, a MixTS M is translated to a KMTS K in two steps. First, every *must only* transition $a \xrightarrow{\text{must}} b$ of M is replaced by a pair of *must* and *may* transitions $a \xrightarrow{\text{must}} \widehat{a \to b}$ and $a \xrightarrow{\text{may}} \widehat{a \to b}$, where $\widehat{a \to b}$ is a (possibly new) abstract state such that $\gamma(\widehat{a \to b}) = \gamma(b) \cap \gamma(R_M^{\text{may}}(a))$. Second, *may* and *must* transitions are added for all states introduced in the first step. We formalize this below.

Definition 5 (MTOK). *Let $M = \langle S_M, R_M^{\text{may}}, R_M^{\text{must}} \rangle$ be a MixTS. The translation* $\text{MTOK}(M)$ *is a KMTS $K = \langle S_K, R_K^{\text{may}}, R_K^{\text{must}} \rangle$, s.t.*

$$S_K \triangleq S_M \cup S^+ \quad S^+ \triangleq \{\widehat{a \to b} \mid \exists (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \cdot \forall s \in S_M \cdot \gamma(s) \neq \gamma(\widehat{a \to b})\}$$
$$R_K^{\text{may}} \triangleq R_M^{\text{may}} \cup \text{REPL} \cup \text{IMAY} \cup \text{IMO}$$
$$R_K^{\text{must}} \triangleq (R_M^{\text{must}} \cap R_M^{\text{may}}) \cup \text{REPL} \cup \text{IMUST} \cup \text{IMO},$$

where

$$\text{REPL} \triangleq \{(a, \widehat{a \to b}) \mid \exists (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}})\}$$
$$\text{IMAY} \triangleq \{(\widehat{a \to b}, b') \mid \exists a, b, b' \in S_M \cdot (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \wedge (b,b') \in R_M^{\text{may}} \wedge \widehat{a \to b} \in S^+\}$$
$$\text{IMUST} \triangleq \{(\widehat{a \to b}, b') \mid \exists a, b, b' \in S_M \cdot (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \wedge (b,b') \in (R_M^{\text{must}} \cap R_M^{\text{may}}) \wedge \widehat{a \to b} \in S^+\}$$
$$\text{IMO} \triangleq \{(\widehat{a \to b}, \widehat{b \to b'} \mid \exists a, b, b' \in S_M \cdot (a,b), (b,b') \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \wedge \widehat{a \to b} \in S^+\}$$

In Definition 5, REPL denotes transitions that replace *must only* transitions, and IMAY, IMUST and IMO denote transitions from newly added states in S^+ that correspond to *may*, *must*, and *must only* transitions of the original system, respectively. In our example of $\text{MTOK}(M_3)$, we have $S^+ = \{a_5\}, \text{REPL} = \{(a_1, a_5), (a_2, a_4)\}, \text{IMUST} = \emptyset, \text{IMO} = \{(a_5, a_4)\}$, and $\text{IMAY} = \{(a_5, a_1), (a_5, a_2), (a_5, a_3)\}$. The result of the translation MTOK is a KMTS: every *must* transition is matched by a *may* transition.

Theorem 3. *Let M be a MixTS, and $K = \text{MTOK}(M)$. Then K is a KMTS, and M and K are semantically equivalent.*

A corollary of Theorem 3 is that MixTSs and KMTSs are equivalent w.r.t. thorough semantics. Let L_M be a labeling function for M. We extend MTOK to $\langle M, L_M \rangle$ such that $\text{MTOK}(\langle M, L_M \rangle) \triangleq \langle K, L_K \rangle$, where $K = \text{MTOK}(M)$, and L_K is a labeling function for S_K defined as follows:

$$L_K(a) \triangleq \begin{cases} L_M(a) & \text{if } a \in S_M \\ \bigcup_{\{s \in S_M \mid \gamma(a) \subseteq \gamma(s)\}} L_M(s) & \text{if } a \in S^+ \end{cases}$$

Then, L_K is well-defined and approximates the same labellings as L_M. This is sufficient to ensure that $\langle M, L_M \rangle$ and $\langle K, L_K \rangle$ satisfy the same properties under thorough semantics.

Corollary 2. *Let $\langle M, L_M \rangle$ be a MixTS model and $\langle K, L_K \rangle = \text{MTOK}(\langle M, L_M \rangle)$. Then, $\langle M, L_M \rangle$ and $\langle K, L_K \rangle$ are equivalent w.r.t. thorough semantics.*

Complexity. Let $M = \langle S_M, R_M^{\text{may}}, R_M^{\text{must}} \rangle$ be a MixTS, and K be a KMTS such that $K = \text{MTOK}(M)$. The size of M is bounded by $O(|S_M \times S_M|)$. In the worst case, the translation adds a new state for each *must only* transition in $R_M^{\text{must}} \setminus R_M^{\text{may}}$. Therefore, the number of new states $|S^+|$ is bounded by $|S_M \times S_M|$, and $|K|$ is bounded by $O(|S_M \times S_M|^2)$.

MixTSs are more succinct than KMTSs: for a fixed statespace S, the set of MixTSs over S is strictly more expressive than the set of KMTSs over S. This holds because for every state t added by MTOK, there exists a subset $U \subseteq S$ s.t. $\gamma(t) = \gamma(U)$.

4 Reduced Inductive Semantics

GKMTSs and MixTSs are equally expressive: a GKMTS model and its equivalent MixTS model satisfy the same properties under thorough semantics. However, thorough model-checking is expensive. In practice, model-checking of partial models is done w.r.t. a more tractable inductive semantics SIS. GKMTSs are more precise than MixTSs w.r.t. SIS: for any $\varphi \in L_\mu$, model-checking φ in a GKMTS model $\mathcal{G}$ w.r.t. SIS is more precise than model-checking it in the MixTS model $\mathcal{M} = \text{GTOM}(\mathcal{G})$. However, the direct use of GKMTSs in symbolic model checkers has been hampered by the difficulty of encoding hyper-transitions into BDDs. In this section, we propose a new semantics, called *reduced inductive semantics* (RIS), that is inductive while being strictly more precise than SIS. We show that GKMTSs and MixTSs are equivalent w.r.t. RIS. In the next section, we provide an efficient symbolic model checking procedure for computing RIS over MixTSs. The outcome is an algorithm that combines the benefits of the efficient symbolic encoding of MixTSs with the model-checking precision of GKMTSs.

In Sec. 4.1, we illustrate the differences between GKMTSs and MixTSs w.r.t. SIS; define RIS in Sec. 4.2; and show how to effectively model-check w.r.t. RIS in Sec. 4.3.

4.1 Example

Let p and q denote predicates $(x > 0)$ and $odd(x)$, respectively. Consider the model $\mathcal{G}_1 = \langle G_1, L_{G_1} \rangle$, where G_1 is shown in Fig. 1, and L_{G_1} is a labeling function that labels each abstract state as shown in Fig. 1. Let $\mathcal{M}_1 = \langle M_1, L_{M_1} \rangle$ be the model obtained from $\mathcal{G}_1$ by GTOM, where M_1 is shown in Fig. 1 and $L_{M_1}(s) \triangleq$ **if** $s = a_5$ **then** $\{p\}$ **else** $L_{G_1}(s)$.

Compare the value of $\varphi \triangleq \Diamond(q \vee \neg q)$ under SIS on $\mathcal{G}_1$ and $\mathcal{M}_1$:

$$\|\varphi\|_c^{\mathcal{G}_1} = \langle\{a_1, a_2, a_3\}, \{a_1, a_2, a_3, a_4\}\rangle \qquad \|\varphi\|_c^{\mathcal{M}_1} = \langle\{a_2, a_3\}, \{a_1, a_2, a_3, a_4, a_5\}\rangle$$

According to $\mathcal{G}_1$, in all states corresponding to a_1, φ is true . According to $\mathcal{M}_1$, the value of φ is unknown in exactly the same states. Since $\mathcal{M}_1 = \text{GTOM}(\mathcal{G}_1), \mathcal{G}_1 \equiv_a \mathcal{M}_1$. Thus, although $\mathcal{M}_1$ and $\mathcal{G}_1$ are semantically equivalent, $\mathcal{M}_1$ is less precise than $\mathcal{G}_1$ for model-checking w.r.t. SIS.

Let us reexamine the above example. First, there is no precision loss during the evaluation of $q \vee \neg q$:

$$\begin{aligned} e_1 &= \|q \vee \neg q\|_c^{\mathcal{G}_1} = \langle\{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_4\}\rangle \qquad (\star) \\ e_2 &= \|q \vee \neg q\|_c^{\mathcal{M}_1} = \langle\{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_4, a_5\}\rangle \end{aligned}$$

Since $\gamma(\mathsf{U}(e_1)) = \gamma(\mathsf{U}(e_2))$ and $\gamma(\overline{\mathsf{O}(e_1)}) = \gamma(\overline{\mathsf{O}(e_2)}) = \gamma(\emptyset)$, $e_1 \equiv_a e_2$. However, there is a subtle difference between e_1 and e_2. In state a_5 of M_1, $q \vee \neg q$ is unknown even though it is true in both a_2 and a_3, and $\gamma(a_5) = \gamma(a_2) \cup \gamma(a_3)$. This minor imprecision is then magnified by the $\Diamond$ operator.

This loss of precision is not limited to tautologies. For example, $\mu Z \cdot (\neg p \wedge q) \vee \Diamond Z$, i.e, $EF(\neg p \wedge q)$ in CTL, is true in state a_1 of $\mathcal{G}_1$, but is unknown in a_1 of $\mathcal{M}_1$.

4.2 Reduced Inductive Semantics for Partial Models

In this section, we define the reduced inductive semantics (RIS). The new semantics is inductive and is *strictly more precise* than SIS. The key idea is to eliminate any local imprecision by using a special *reduction* operator.

Let S be an abstract statespace, and $e, e' \in 2^S \times 2^S$ be two abstract elements. Recall that in the information order e is less than e', i.e., $e \preceq_i e'$, if $\mathsf{U}(e)$ is contained in $\mathsf{U}(e')$, and $\mathsf{O}(e)$ contains $\mathsf{O}(e')$. We define the *reduction* operator as follows: $\texttt{RED}(e) \triangleq \langle \texttt{RED}_{\mathsf{U}}(U), \texttt{RED}_{\mathsf{O}}(O)\rangle$, where $\texttt{RED}_{\mathsf{U}}(U) \triangleq \{s \mid \gamma(s) \subseteq \gamma(U)\}$, and $\texttt{RED}_{\mathsf{O}}(O) \triangleq \{s \mid \gamma(s) \nsubseteq \gamma(\overline{O})\}$. Intuitively, $\texttt{RED}(e)$ increases $\mathsf{U}(e)$ and decreases $\mathsf{O}(e)$ as much as possible without affecting the semantic meaning of e. That is, $\texttt{RED}(e)$ is the largest element w.r.t. information ordering that is semantically equivalent to e. For example, consider $\texttt{RED}(e_2)$, where e_2 is as defined by $(\star)$ above. Then,

$$e_3 = \texttt{RED}(e_2) = \langle\{a_1, a_2, a_3, a_4, a_5\}, \{a_1, a_2, a_3, a_4, a_5\}\rangle \qquad (\star\star)$$

e_3 differs from e_2 only in the addition of a_5 to $\mathsf{U}(e_3)$. Since $\gamma(\mathsf{U}(e_2)) = \gamma(\mathsf{U}(e_3))$ and $\gamma(\overline{\mathsf{O}(e_2)}) = \gamma(\overline{\mathsf{O}(e_3)})$ $e_2 \equiv_a e_3$; but e_3 is more informative since $\mathsf{U}(e_2) \subset \mathsf{U}(e_3)$.

An element $e = \langle U, O\rangle \in 2^S \times 2^S$ is *monotone* iff

$$s_1 \preceq_S s_2 \Rightarrow (s_1 \in U \Rightarrow s_2 \in U \wedge s_1 \notin O \Rightarrow s_2 \notin O)$$

$\texttt{RED}(e)$ is monotone for any e, and commutes with propositional operations on monotone elements. That is, let e and e' be monotone elements of $2^S \times 2^S$. Then, $\sim e \equiv_a \sim\texttt{RED}(e)$, and $e \sqcap e' \equiv_a \texttt{RED}(e) \sqcap \texttt{RED}(e')$.

RIS is defined by applying the RED operator before and after $\Diamond$ to prevent it from propagating imprecision.

Definition 6 (RIS). *Let $\mathcal{M} = \langle M, L_M\rangle$ be a model, s.t. $M = \langle S, R^{\text{may}}, R^{\text{must}}\rangle$ and $\sigma : Var \to 2^S \times 2^S$. The* reduced inductive semantics *of $\varphi \in L_\mu$ is defined as follows:*

$$||p||^{\mathcal{M}}_{r,\sigma} \triangleq \langle \{s \mid p \in L_M(s)\}, \{s \mid \neg p \notin L_M(s)\}\rangle$$
$$||\neg\varphi||^{\mathcal{M}}_{r,\sigma} \triangleq \sim||\varphi||^{\mathcal{M}}_{r,\sigma} \qquad ||\varphi \wedge \psi||^{\mathcal{M}}_{r,\sigma} \triangleq ||\varphi||^{\mathcal{M}}_{r,\sigma} \sqcap ||\psi||^{\mathcal{M}}_{r,\sigma} \qquad ||Z||^{\mathcal{M}}_{r,\sigma} \triangleq \sigma(Z)$$
$$||\Diamond\varphi||^{\mathcal{M}}_{r,\sigma} \triangleq \mathtt{RED}(\langle pre_{\mathsf{U}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||^{\mathcal{M}}_{r,\sigma}))), pre_{\mathsf{O}}(\mathtt{RED}_{\mathsf{O}}(\mathsf{O}(||\varphi||^{\mathcal{M}}_{r,\sigma})))\rangle)$$
$$||\mu Z \cdot \varphi||^{\mathcal{M}}_{r,\sigma} \triangleq \langle lfp^{\sqsubseteq}\left(\lambda Q \cdot \mathsf{U}(||\varphi||^{\mathcal{M}}_{r,\sigma[Z\mapsto Q]})\right), lfp^{\sqsubseteq}\left(\lambda Q \cdot \mathsf{O}(||\varphi||^{\mathcal{M}}_{r,\sigma[Z\mapsto Q]})\right)\rangle$$

The only difference between RIS (Definition 6) and SIS (Definition 2) is the semantics of $\Diamond$. Since we assume that state-labellings are monotone, applying RED to other operators as well does not improve precision.

Returning to our running example, RIS of φ on $\mathcal{M}_1$ is computed as follows: RIS of q, $\neg q$, and $q \vee \neg q$ is the same as SIS. Thus, $\|q \vee \neg q\|^{\mathcal{M}_1}_r = e_2$. To compute $\Diamond$, recall from ($\star\star$) that $\mathtt{RED}(e_2) = e_3$; thus, $\|\varphi\|^{\mathcal{M}_1}_r = \langle\{a_1, a_2, a_3, a_5\}, \{a_1, a_2, a_3, a_4, a_5\}\rangle$. Hence, $\|\varphi\|^{\mathcal{M}_1}_r$ is more precise than $\|\varphi\|^{\mathcal{M}_1}_c$.

Theorem 4. *RIS is more precise than SIS:* $\|\varphi\|_c \preceq_a \|\varphi\|_r$.

The previous example illustrates another important point: GKMTSs and MixTSs are equivalent w.r.t. RIS. For example, $\|\varphi\|^{\mathcal{M}_1}_r$ is equivalent to $\|\varphi\|^{\mathcal{G}_1}_r$. The following theorem formalizes this.

Theorem 5. *Let $\mathcal{G}$ be a GKMTS, and $\mathcal{M} = \textsc{GtoM}(\mathcal{G})$. Then, $\mathcal{G}$ and $\mathcal{M}$ are equivalent w.r.t. RIS:* $\forall\varphi \in L_\mu \cdot \|\varphi\|^{\mathcal{G}}_r \equiv_a \|\varphi\|^{\mathcal{M}}_r$

Our new semantics RIS is both inductive and precise enough to make GKMTSs and MixTSs equivalent. However, the definition of RED operator is based on concretization, γ, of abstract elements. In practice, reasoning directly about concrete elements may be undecidable or inefficient. We address this limitation next.

4.3 Reduced Inductive Semantics for Monotone Models

We study the reduction operator RED of RIS in the context of monotone models. *Monotone models*, formally defined below, are as expressive as their regular counterparts [11]: for any model there exists an equivalent monotone one. The monotonicity condition simply ensures that all information that can be derived from existing *may* and *must* transitions is made explicit in the model. Furthermore, models built by automated predicate abstraction [10] are monotone by construction. Thus, restricting RED to monotone models is neither a theoretical nor a practical restriction.

Definition 7. *A MixTS $M = \langle S, R^{\text{may}}, R^{\text{must}}\rangle$ is* monotone *iff for any $s_1 \preceq_S s_2, t_2 \preceq_S t_1$,*

$$((s_2, t_2) \in R^{\text{may}} \Rightarrow (s_1, t_1) \in R^{\text{may}}) \wedge \left((s_1, t_1) \in R^{\text{must}} \Rightarrow (s_2, t_2) \in R^{\text{must}}\right)$$

A model $\mathcal{M} = \langle M, L_M\rangle$ is monotone *iff the MixTS M is monotone.*

For monotone models, RED can be computed effectively, as we show below.

For a state $s \in S$, the *upset* of s is defined as $\uparrow s \triangleq \{t \in \alpha[S] \mid s \preceq_a t\}$. Thus, $\uparrow s$ is the set of all those states in $\alpha[S]$ that are more precise than s. For example, let S_1 be the statespace of M_1 in Fig. 1. Then, $\alpha[S_1] = \{a_1, a_2, a_3, a_4\}$, and $\uparrow a_5 = \{a_2, a_3\}$. Note that the state s and the set $\uparrow s$ approximate the same set of concrete states, i.e., $\gamma(s) = \gamma(\uparrow s)$. For example, $\gamma(\uparrow a_5) = \gamma(a_5) = (x > 0)$.

Let $e = \langle U, O\rangle$ be a monotone element of $2^S \times 2^S$, and $s \in S$. By monotonicity, $\gamma(s) \subseteq \gamma(U)$ iff $\uparrow s \subseteq U$. Dually, $\gamma(s) \not\subseteq \gamma(\overline{O})$ iff $\uparrow s \not\subseteq \overline{O}$. We define a new operator $\texttt{red}$ as follows: $\texttt{red}(e) \triangleq \langle \texttt{red}_{\texttt{U}}(U), \texttt{red}_{\texttt{O}}(O)\rangle$, where $\texttt{red}_{\texttt{U}}(U) \triangleq \{s \mid \uparrow s \subseteq U\}$, and $\texttt{red}_{\texttt{O}}(O) \triangleq \{s \mid \uparrow s \not\subseteq \overline{O})\}$.

Theorem 6. *Let S be an abstract statespace, and e be a monotone element in $2^S \times 2^S$. Then, $\texttt{red}(e) = \texttt{RED}(e)$.*

$\texttt{red}$ can be computed effectively since it does not reason about concrete elements directly.

In this section, we have introduced a new inductive semantics RIS, shown that it is more precise than SIS, and that GKMTSs and MixTSs are equivalent w.r.t. RIS. RIS can be computed effectively on monotone models, which is not a limitation since monotone models are as expressive as their non-monotone counterparts.

5 Symbolic Model-Checking of RIS Using BDDs

In this section, we describe a symbolic algorithm RIS that implements the RIS semantics for monotone models constructed using predicate abstraction. These are the models used by existing software model-checkers [12].

Our implementation is based on the following observation. Let S be an abstract statespace. Then, for any monotone element of $2^S \times 2^S$ there exists a semantically equivalent element in $2^{\alpha[S]} \times 2^{\alpha[S]}$.

Theorem 7. *Let $e_1 = \langle U_1, O_1\rangle$ be a monotone element of $2^S \times 2^S$, and $e_2 = \langle U_2, O_2\rangle$ be in $2^{\alpha[S]} \times 2^{\alpha[S]}$. If $U_1 \cap \alpha[S] = U_2$ and $O_1 \cap \alpha[S] = O_2$, then $e_1 \equiv_a e_2$.*

This theorem allows us to restrict the algorithm to sets over $\alpha[S]$ instead of sets over S. Another consequence of Theorem 7 is that the transition relations can be simplified as well, since we only need the result of the pre-image in the states of $\alpha[S]$.

```
 1: global var Rmay, Rmust : BDD
 2: func RIS(Expr φ) : BDD
 3:   match φ with
 4:     ATOMIC(p) : return ABSV(BDDVAR("p"),
                               BDDVAR("p"))
 5:     ¬ψ : return ABSNOT(RIS(ψ))
 6:     ψ1 ∧ ψ2 : return ABSAND(RIS(ψ1), RIS(ψ2))
 7:     ψ1 ∨ ψ2 : return ABSOR(RIS(ψ1), RIS(ψ2))
 8:     ◇ψ : return ABSPRE(Rmay, Rmust, RIS(ψ))
 9:     μψ : return RIS_lfp(ψ)
10:     νψ : return RIS_gfp(ψ)
11:
12: func ABSV(BDD u, BDD o) : BDD
13:   sel := BDDVAR("sel")
14:   return BDDITE(sel, u, o)
15:
16: func ABSO(BDD v) = v[0/sel]
17: func ABSU(BDD v) = v[1/sel]
18: func ABSAND(BDD v1, BDD v2) = BDDAND(v1, v2)
19: func ABSOR(BDD v1, BDD v2) = BDDOR(v1, v2)
20: func ABSEQ(BDD v1, BDD v2) = BDDEQ(v1, v2)
21:
22: func ABSNOT(BDD v) : BDD
23:   o := ABSO(v), u := ABSU(v)
24:   return ABSV(BDDNOT(o), BDDNOT(u))
25:
26: func ABSREDU(BDD v) : BDD
27:   if (BDDISCONST(v)) return v
28:   b := BDDROOTVAR(v), h := UVAR(b)
29:   T := ABSREDU(v[1/b]), F := ABSREDU(v[0/b])
30:   tmp := BDDITE(b, T, F)
31:   return BDDITE(h, BDDAND(T, F), tmp)
32:
33: func ABSPRE(BDD Rmay, BDD Rmust, BDD v) : BDD
34:   o := ABSO(v), u := ABSREDU(ABSU(v))
35:   return ABSV(BDDPRE(Rmust, u), BDDPRE(Rmay, o))
```

Fig. 2. The RIS algorithm and its supporting functions

Theorem 8. *Let* $R^{\text{may}} \subseteq S \times S$ *and* $R^{\text{must}} \subseteq S \times S$ *be the* may *and* must *transition relations of a monotone MixTS, respectively, and* $e = \langle U, O\rangle$ *be a monotone element of* $2^S \times 2^S$. *Define* $\hat{U} \triangleq U \cap \alpha[S]$, $\hat{O} \triangleq O \cap \alpha[S]$, $\hat{R}^{\text{must}} \triangleq R^{\text{must}} \cap (\alpha[S] \times S)$, *and* $\hat{R}^{\text{may}} \triangleq R^{\text{may}} \cap (\alpha[S] \times \alpha[S])$. *Then,*

$$\langle pre[R^{\text{must}}](\texttt{RED}_{\texttt{U}}(U)), pre[R^{\text{may}}](\texttt{RED}_{\texttt{O}}(O))\rangle \equiv_a \langle pre[\hat{R}^{\text{must}}](\texttt{RED}_{\texttt{U}}(\hat{U})), pre[\hat{R}^{\text{may}}](\hat{O})\rangle$$

The algorithm RIS is shown in Fig. 2. It uses BDDs to symbolically represent and manipulate sets of states and transition relations. Functions that are prefixed with "BDD" are the standard BDD operations. The algorithm works recursively on the structure of the input formula φ. The fixpoints are computed in the usual way, by iterating until convergence. We describe the details of the implementation below.

Let $P = \{p_1, \ldots, p_n\}$ be a set of n predicates. Recall that $\text{Mon}(P)$ denotes the set of monomials over P, and $\text{MT}(P)$ — the set of minterms over P. Furthermore, $\alpha[\text{Mon}(P)] = \text{MT}(P)$. The input to the algorithm is a MixTS model $\langle M, L_M\rangle$, s.t. $M = (S, R^{\text{may}}, R^{\text{must}})$, $S = \text{Mon}(P)$, and $L_M(s) = Lit(s)$, and an L_μ property φ. Without loss of generality, by Theorem 8, we assume that the transition relations are restricted s.t. $R^{\text{may}} \subseteq \text{MT}(P) \times \text{MT}(P)$, and $R^{\text{must}} \subseteq \text{MT}(P) \times \text{Mon}(P)$.

The algorithm uses the following sets of BDD variables: $B = \{b_i \mid p_i \in P\}$ – the current state Boolean variables, $B' = \{b'_i \mid b_i \in B\}$ – the next state Boolean variables, $H = \{h_i \mid p_i \in P\}$ – the current state unknown variables, and $H' = \{h'_i \mid h_i \in H\}$ – the next state unknown variables. In what follows, we do not distinguish between the BDDs and the corresponding propositional formulas.

A set of minterms $X \subseteq \text{MT}(P)$ is encoded by a propositional formula over B, as usual. For example, let $P = \{p_1, p_2, p_3\}$. Then $b_1 \wedge \neg b_2$ encodes the set $\{p_1 \wedge \neg p_2 \wedge p_3, p_1 \wedge \neg p_2 \wedge \neg p_3\}$. A set of monomials $X \subseteq \text{Mon}(P)$ is encoded by a formula over $B \cup H$. Intuitively, for a monomial m, a variable h_i indicates whether p_i is present in m, and a variable b_i specifies the polarity of the occurrence. Formally, the encoding is

$$\bigvee_{m \in X} \left(\Big(\bigwedge_{p_i \in Lit(m)} \neg h_i \wedge b_i \Big) \wedge \Big(\bigwedge_{\neg p_i \in Lit(m)} \neg h_i \wedge \neg b_i \Big) \wedge \Big(\bigwedge_{p_i \in P \setminus Term(m)} h_i \Big) \right)$$

For example, $(\neg h_1 \wedge b_1) \wedge (\neg h_2 \wedge \neg b_2) \wedge h_3$ represents a singleton set $\{p_1 \wedge \neg p_2\}$.

An abstract value $e = \langle U, O\rangle$ is encoded in a single BDD by a formula $(\texttt{sel} \wedge U) \vee (\neg\texttt{sel} \wedge O)$, where sel is a designated BDD variable. This encoding is implemented by function ABSV. The U and O elements of value e are extracted using ABSU and ABSO, respectively. Abstract intersection (ABSAND), union (ABSOR), and equality (ABSEQ) are done using the corresponding BDD operations. Abstract negation (ABSNOT) is implemented following its definition in Sec. 2.

The may transition relation $R^{\text{may}} \subseteq \text{MT}(P) \times \text{MT}(P)$ is encoded by a formula over $B \cup B'$ as usual. Similarly, the must relation $R^{\text{must}} \subseteq \text{MT}(P) \times \text{Mon}(P)$ is encoded by a formula over $B \cup B' \cup H'$, where the primed variables are used to encode the destination state. For example, a *must* transition from a state $(p_1 \wedge p_2 \wedge p_3)$ to a state $(p_1 \wedge \neg p_2)$ is represented by $(b_1 \wedge b_2 \wedge b_3) \wedge ((\neg h'_1 \wedge b'_1) \wedge (\neg h'_2 \wedge \neg b'_2) \wedge h'_3)$.

Function ABSREDU implements the $\texttt{red}_{\texttt{U}}$ reduction operator of Sec. 4.3 using the following observation: let $Q \subseteq \text{Mon}(P)$ be a monotone subset, and $a \in \text{Mon}(P)$. If $a \in \text{MT}(P)$, then $\uparrow a \subseteq Q \Leftrightarrow a \in Q$; otherwise, $\uparrow a \subseteq Q$ iff $\uparrow(a \wedge p) \subseteq Q$ and $\uparrow(a \wedge \neg p) \subseteq Q$, where p is a term not occurring in a. ABSREDU applies this reasoning

recursively on the input diagram, using function UVAR to find a variable $h_i \in H$ for each variable $b_i \in B$. Function ABSPRE implements the pre-image computation based on Theorem 8.

Theorem 9. *For a monotone MixTS $\mathcal{M}$ and $\varphi \in L_\mu$, algorithm* $\mathrm{RIS}(\varphi)$ *in Fig. 2 returns the symbolic representation of* $\|\varphi\|_r^{\mathcal{M}}$.

The main difference between the symbolic implementations of SIS and our RIS is the extra ABSREDU operation in function ABSPRE (line 29 in Fig. 2). ABSREDU is similar to existential quantification (BDDEXISTS) of BDDs, with one exception: BDDEXISTS uses BDDOR in each iteration, but ABSREDU uses one BDDAND and two BDDITE operations. Thus, ABSREDU has the same complexity as BDDEXISTS, and symbolic implementations of RIS and SIS also have the same complexity. This means that the extra precision of RIS comes "for free", without penalty in complexity.

6 Experiments

To empirically evaluate the cost and performance of RIS versus SIS, we have implemented symbolic algorithms for computing both of them using CUDD [21] library, and analyzed reachability and non-termination properties over a realistic model. While our algorithm in Fig. 2 can analyze any μ-calculus formula, our experiments considered just reachability and non-termination properties because of their practical interest.

For the model, we used a template program built out of n blocks, each based on an example from [19] and having one integer variable. The method of [10] was applied to build an abstract MixTS via predicate abstraction. We checked one reachability (least fixed-point) property, $\mathtt{Prop}_1$, and two non-termination (greatest fixed-point) properties, $\mathtt{Prop}_2$, and $\mathtt{Prop}_3$, computing the standard and reduced semantics of the properties. In both cases, we measured the size of the abstract models using the number of BDD nodes, the total analysis time, the number of iterations of the fixpoint computation, and the time spent in the ABSREDU operation for RIS. To compare the precision of the results, we considered two sets of initial states, $\mathtt{I}_1$ and $\mathtt{I}_2$, and checked whether conclusive results can be obtained over them.

The results are summarized in Fig. 3. The top part of the table shows that RIS models enjoy significantly smaller encodings than their SIS counterparts, due to restricted transition relations (see Theorem 8). RIS is more precise than SIS: for the two sets of initial states, RIS produces conclusive results for both of them w.r.t. the three properties being checked, whereas SIS cannot decide whether $\mathtt{Prop}_1$ and $\mathtt{Prop}_2$ hold in $\mathtt{I}_2$. As expected, the extra precision of RIS does not cause a complexity penalty: the experiments show that the increases of the analysis time w.r.t. the size of the models for both RIS and SIS are comparable. In all of the cases, the time spent in ABSREDU, which represents the main difference between the two semantics, comprises roughly 20% - 25% of the total time.

Note that RIS and SIS may require different numbers of iterations of fixpoint computation depending on the structure of the models and the type of the fixpoint computed. For example, RIS required more iterations than SIS for $\mathtt{Prop}_1$, whereas it converged in just two steps (vs. the number of iterations proportional to the model size for SIS) for $\mathtt{Prop}_2$ over the same model.

	n	SIS				RIS				
Model Size	100	370,070				216,689				
	200	1,460,270				853,389				
	250	2,275,196				1,329,215				
Prop.	n	Analysis (sec.)	Iter.	$\mathtt{I}_1$	$\mathtt{I}_2$	Analysis (sec.)	ABSREDU (sec.)	Iter.	$\mathtt{I}_1$	$\mathtt{I}_2$
$\mathtt{Prop}_1$	100	2.20	301	T	U	3.60	0.74	401	T	T
	200	15.36	601			27.77	6.45	801		
	250	28.92	751			55.19	13.40	1001		
$\mathtt{Prop}_2$	100	3.60	203	T	U	0.03	$< 10^{-4}$	2	T	T
	200	27.16	403			0.12	$< 10^{-4}$	2		
	250	54.62	503			0.19	$< 10^{-4}$	2		
$\mathtt{Prop}_3$	100	33.96	400	F	F	21.24	4.5	400	F	F
	200	395.24	800			258.72	42.44	800		
	250	1108.67	1000			546.88	101.20	1000		

Fig. 3. Experimental results (T, F and U denote *True*, *False* and *Unknown*, respectively)

These experiments suggest that using the more precise RIS semantics improves the overall performance of model checking, making it a viable alternative to SIS in practice.

7 Related Work and Conclusion

Godefroid and Jagadeesan [8], and Gurfinkel and Chechik [9] proved that the models in the KMTS family have the same expressive power and are equally precise for SIS. Dams and Namjoshi [5] showed that the three families considered in this paper are subsumed by tree automata. We complete the picture by proving that the three families are equivalent as well. Specifically, we showed that KMTSs, MixTSs and GKMTSs are relatively complete (in the sense of [5]) with one another.

We did not consider Hyper TSs (HTSs) [20] which allow for both *must* and *may* hyper-transitions. As pointed out in [20], *may* hyper-transitions can be eliminated by increasing the abstract statespace, making HTSs exactly as expressive as GKMTSs.

Both GKMTSs and MixTSs support monotonic abstraction refinement under SIS [4, 11, 19]. The same result holds under RIS, because for any two partial model M and M' over the same abstract statespace, if M' is less precise than M under SIS, i.e., $\forall \varphi \in L_\mu \cdot \|\varphi\|_c^{M'} \preceq_a \|\varphi\|_c^M$, M' is also less precise than M under RIS.

Our reduction operator RED is an instance of normalization from Abstract Interpretation [3] that is sometimes used to provide a canonical representation of equivalent abstract properties. Our symbolic implementation ABSREDU is similar to the semantic minimization of 3-valued propositional formulas [18].

Since RIS is inductive, its precision lies between SIS and thorough semantics. Thus, existing results comparing SIS and thorough semantics, e.g., [7, 9, 17], apply to RIS as well. We leave open the question of whether the additional precision enjoyed by RIS can be used to improve the above results.

In this paper, we compared three families of partial modeling formalisms: KMTSs, MixTSs and GKMTSs. We showed that they are equally expressive – a model from

one formalism can be transformed into a semantically equivalent model from the other. Thus, neither hyper-transitions nor restrictions on *may* and *must* transitions affect expressiveness. They do, of course, affect the succinctness of the models.

We further introduced a new inductive semantics, RIS, for partial models. RIS is more precise than SIS, making MixTSs and GKMTSs equivalent w.r.t. model-checking. We also described a symbolic implementation of model-checking w.r.t. RIS. The outcome is an algorithm that combines the efficient symbolic encoding of MixTSs with the model-checking precision of GKMTSs. The symbolic algorithm was evaluated empirically. Our experiments suggest that RIS should be a good alternative to SIS for predicate abstraction-based model-checkers. We leave further investigations along this research direction to future work.

Acknowledgments. We thank Sagar Chaki, Orna Grumberg, and Yael Meller for comments on the paper; and Laurie Lugrin for her help with the experiments.

References

1. Bruns, G., Godefroid, P.: Generalized Model Checking: Reasoning about Partial State Spaces. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 168–182. Springer, Heidelberg (2000)
2. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-Valued Symbolic Model-Checking. ACM TOSEM 12(4), 1–38 (2003)
3. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. J. of Logic and Computation 2(4), 511–547 (1992)
4. Dams, D., Gerth, R., Grumberg, O.: Abstract Interpretation of Reactive Systems. ACM TOPLAS 2(19), 253–291 (1997)
5. Dams, D., Namjoshi, K.S.: Automata as Abstractions. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 216–232. Springer, Heidelberg (2005)
6. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-Valued Abstractions of Games: Uncertainty, but with Precision. In: LICS, pp. 170–179 (2004)
7. Godefroid, P., Huth, M.: Model Checking v.s. Generalized Model Checking: Semantic Minimizations for Temporal Logics. In: LICS, pp. 158–167 (2005)
8. Godefroid, P., Jagadeesan, R.: On the Expressiveness of 3-Valued Models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 206–222. Springer, Heidelberg (2002)
9. Gurfinkel, A., Chechik, M.: How Thorough Is Thorough Enough? In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 65–80. Springer, Heidelberg (2005)
10. Gurfinkel, A., Chechik, M.: Why Waste a Perfectly Good Abstraction? In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)
11. Gurfinkel, A., Wei, O., Chechik, M.: Systematic Construction of Abstractions for Model-Checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 381–397. Springer, Heidelberg (2005)
12. Gurfinkel, A., Wei, O., Chechik, M.: YASM: A Software Model-Checker for Verification and Refutation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 170–174. Springer, Heidelberg (2006)
13. 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, pp. 155–169. Springer, Heidelberg (2001)

14. Kozen, D.: Results on the Propositional μ-calculus. Theoretical Computer Science 27, 334–354 (1983)
15. Larsen, K., Thomsen, B.: A Modal Process Logic. In: LICS, pp. 203–210 (1988)
16. Larsen, P.: The Expressive Power of Implicit Specifications. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 204–216. Springer, Heidelberg (1991)
17. Nejati, S., Gheorghiu, M., Chechik, M.: Thorough Checking Revisited. In: FMCAD, pp. 106–116 (2006)
18. Reps, T.W., Loginov, A., Sagiv, S.: Semantic Minimization of 3-Valued Propositional Formulae. In: LICS, pp. 40–54 (2002)
19. Shoham, S., Grumberg, O.: Monotonic Abstraction-Refinement for CTL. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 546–560. Springer, Heidelberg (2004)
20. Shoham, S., Grumberg, O.: 3-Valued Abstraction: More Precision at Less Cost. In: LICS, pp. 399–410 (2006)
21. Somenzi, F.: CUDD: CU Decision Diagram Package Release (2001)

Counterexample Generation for Discrete-Time Markov Chains Using Bounded Model Checking*

Ralf Wimmer, Bettina Braitling, and Bernd Becker

Chair of Computer Architecture
Albert-Ludwigs-University Freiburg im Breisgau, Germany
{wimmer,braitlin,becker}@informatik.uni-freiburg.de

Abstract. Since its introduction in 1999, bounded model checking has gained industrial relevance for detecting errors in digital and hybrid systems. One of the main reasons for this is that it always provides a counterexample when an erroneous execution trace is found. Such a counterexample can guide the designer while debugging the system.

In this paper we are investigating how bounded model checking can be applied to generate counterexamples for a different kind of model—namely discrete-time Markov chains. Since in this case counterexamples in general do not consist of a single path to a safety-critical state, but of a potentially large set of paths, novel optimization techniques like loop-detection are applied not only to speed-up the counterexample computation, but also to reduce the size of the counterexamples significantly. We report on some experiments which demonstrate the practical applicability of our method.

1 Introduction

Nowadays, formal verification plays a crucial role in the design process of digital circuits. In particular model checking, i. e., the proof that a system exhibits a set of properties, which are part of the specification, has gained great importance in industry. The reasons for its success are that it can be performed automatically and—in case a property is violated—that it is often able to generate a counterexample, which guides the designer when debugging the system. By using symbolic methods (e. g. ordered binary decision diagrams, OBDDs [1]) model checking can be applied to fairly large systems. Nevertheless, there are many practically important systems which cannot be verified using OBDDs (e. g., if they contain multipliers), because the size of the OBDD representations may explode.

To overcome this problem, Clarke et al. [2] suggested a method called Bounded Model Checking (BMC). It aims at the *refutation* of invariant properties. The reachability of a state within a fixed number of steps which violates an invariant is thereby formulated as a satisfiability problem in conjunctive normal form (CNF). The advantage is that the size of this CNF is linear in the number

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

N.D. Jones and M. Müller-Olm (Eds.): VMCAI 2009, LNCS 5403, pp. 366–380, 2009.

of gates of the digital circuit. Due to the enormous improvements in solving such satisfiability problems during the last two decades, BMC has successfully been applied to industrial systems.

On the other hand, on a higher modelling level, the designer often has to cope with uncertainties. They can result from unpredictable behavior of the environment, component failures, user interaction, ... One of the most common models for this scenario are discrete-time Markov chains (DTMCs). Model checking, which has been extended to DTMCs, is not as successful as model checking for digital circuits for three reasons: (1) The size of the systems which can be handled by state-of-the-art tools is still quite limited compared to the size of verifiable circuits, (2) all state-of-the-art model checkers use inexact arithmetic for efficiency reasons (i. e., using IEEE 754 floating point arithmetic and iterative solution methods for linear equation systems). This can cause numerical instabilities which themselves may lead to wrong results [3]. (3) The model checker is not able to provide a counterexample for violated properties, since the check is done by solving a linear equation system and not by traversing the state space.

In this paper we tackle the latter two problems. We propose an extension to bounded model checking such that we can use it to certify that invariant properties of the form "The probability to reach a state which violates the invariant is at most p" do not hold. For this purpose we provide counterexamples which can be checked easily for correctness by using exact arithmetic.

Related work. The generation of counterexamples for various kinds of properties has been studied extensively for non-stochastic models (see e. g. [4,5,6,7]). However, little research has been done on counterexamples for stochastic models. Aljazzar et al. [8,9] apply informed search methods like Best First Search or A* to generate counterexamples for continuous-time Markov chains, whereas Han and Katoen [10] propose a different method to obtain counterexamples: They apply a k-shortest-paths algorithm to a Markov chain. Therewith they compute a counterexample consisting of a minimal number of paths. Although Han and Katoen do not provide any experimental results in their paper, we suppose that their method—as well as the method by Aljazzar et al.—does not scale well to large systems due to the explicit state representation it relies on. Further on both approaches do not apply any method to reduce the size of the counterexamples. As our experiments will show, it is crucial to reduce the size of the counterexamples in order to get useful counterexamples in reasonable time. We will overcome both problems by using a symbolic representation and effective techniques to reduce the size of counterexamples via loop detection.

The necessity of reducing the size of counterexamples has also been recognized by Damman et al. [11]. They compute regular expressions describing a sufficiently large set of paths from the initial state to a target state. This can be considered as a generalization of the loop-detection technique we will present later, but is restricted to explicitly represented state spaces.

In the paper [12] by Andrés et al. abstract away the details of strongly connected components of Markov chains by replacing the SCCs by edges that are labeled with the probability to walk through the SCC. On the resulting acyclic

Markov chain, counterexamples are computed which are more compact by collapsing the SCCs, but lack the details how the SCCs can be traversed.

Recently, Hermanns et al. extended counterexample-guided abstraction refinement (CEGAR) to Markov decision processes (MDPs) [13]. In this scenario a counterexample is an adversary which resolves the nondeterminism in each state, such that the probability of reaching a set of states exceeds some bound. For the computation of counterexamples in the abstract system, they apply the shortest-paths algorithm of [10], but they could also use our method.

Organization of this paper. In the next section, we will briefly review the foundations of discrete-time Markov chains and bounded model checking. In Section 3 we will show how to bring together DTMCs and BMC. Its practical applicability will be demonstrated experimentally in Section 4, before we conclude in Section 5.

2 Foundations

In this section we will briefly review the basics of discrete-time Markov chains and bounded model checking for digital systems.

2.1 Discrete-Time Markov Chains and Reachability Properties

Definition 1. *Let AP be a set of atomic propositions. A* discrete-time Markov chain *(DTMC) is a tuple $M = (S, s_I, P, L)$ such that S is a finite, non-empty set of states; $s_I \in S$, the initial state; $P : S \times S \to [0, 1]$, the matrix of the one-step transition probabilities; and $L : S \to 2^{AP}$, a labeling function that assigns each state the set of atomic propositions which hold in that state.*

We require the matrix P to be a stochastic matrix, i. e., $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$. A *path* π is a (finite or infinite) sequence $\pi = s_0 s_1 \dots$ of states such that $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. For a finite path $\pi = s_0 s_1 \dots s_n$, $|\pi| = n$ denotes its length. The i^{th} state of π is denoted by π^i, i. e., $\pi^i = s_i$. Furthermore let $\pi{\uparrow}^i$ be the i^{th} prefix of π ($\pi{\uparrow}^i = s_0 s_1 \dots s_i$). The set of infinite paths starting in state s is denoted by Path_s.

Definition 2. *Let $M = (S, s_I, P, L)$ be a DTMC and $s \in S$. We define a* probability space $\Psi_s = (\text{Path}_s, \Delta_s, \Pr_s)$ *such that*

- *Δ_s is the smallest σ-algebra generated by* Path_s *and the basic cylinders that are subsets of* Path_s*. Thereby, for a finite path π, the basic cylinder over π is defined as $\Delta(\pi) = \{\pi' \in \text{Path}_s \mid \pi'{\uparrow}^{|\pi|} = \pi\}$.*
- *$\Pr_s$ is the uniquely defined probability measure that satisfies the following constraints: $\Pr_s(\text{Path}_s) = 1$ and for all basic cylinders $\Delta(s s_1 s_2 \dots s_n)$ over S:*

$$\Pr_s(\Delta(s s_1 s_2 \dots s_n)) = P(s, s_1) \cdot P(s_1, s_2) \cdot \dots \cdot P(s_{n-1}, s_n).$$

Many investigations can be carried out on even simpler systems, namely Kripke structures, which are obtained by omitting the transition probabilities of the Markov chain:

Definition 3. *Let $M = (S, s_I, P, L)$ be a discrete-time Markov chain. The underlying* Kripke structure *is a labeled graph $G = (S, s_I, T, L)$ such that $(s, s') \in T$ iff $P(s, s') > 0$.*

The properties under consideration are PCTL formulae [14] of the form $\mathcal{P}_{\leq p}(a \mathsf{U} b)$ with $a, b \in AP$, meaning that the probability to walk along a path from the initial state to a state which satisfies b, passing only states in which a holds, is less or equal p. More formally:

Definition 4. *Let π be a path in a Markov chain $M = (S, s_I, P, L)$. It satisfies the formula $a \mathsf{U} b$ (written $\pi \vDash a \mathsf{U} b$) iff*

$$\exists i \geq 0 : \big(b \in L(\pi^i) \wedge \forall\, 0 \leq j < i : a \in L(\pi^j)\big).$$

A state s satisfies the formula $\mathcal{P}_{\leq p}(a \mathsf{U} b)$ (written $s \vDash \mathcal{P}_{\leq p}(a \mathsf{U} b)$) iff

$$\Pr_s\big(\{\pi \in \mathrm{Path}_s \mid \pi \vDash a \mathsf{U} b\}\big) \leq p.$$

If such a property is violated in the initial state s_I, i. e., if the actual probability is greater than p, there is a finite set of finite paths, starting in s_I and satisfying $a \mathsf{U} b$, whose probability measure is greater than p [10].

Definition 5. *Let $M = (S, s_I, P, L)$ be a discrete-time Markov chain for which the property $P_{\leq p}(a \mathsf{U} b)$ is violated in state s_I. A* counterexample *is a set C of paths which start in s_I and satisfy $a \mathsf{U} b$, such that $\Pr_{s_I}(C) > p$.*

Our goal is to provide the user with such a set of paths which testifies that the probability indeed exceeds the bound p.

2.2 Bounded Model Checking for Digital Systems

Bounded model checking [2] is an automatic technique to prove that some invariant is violated in a transition system, i. e., that a state in which the invariant does not hold is reachable from the initial state of the system. The reachability of such a state in a fixed number k of steps is thereby formulated as a satisfiability problem.

Let $I(s)$ be a predicate which is true if s is the initial state and false otherwise. Accordingly, let $P(s)$ be a predicate for the states violating the invariant. The transition relation is encoded by a predicate $T(s, t)$ which is true iff there is a transition from s to t. The existence of a run of length k starting in an initial state and ending in a state in which the invariant does not hold can then be described by the following formula:

$$BMC(k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge P(s_k). \tag{1}$$

If this formula is unsatisfiable for the current unrolling depth k, there is no path of length k from an initial state to a safety-critical state. Otherwise we get a satisfying assignment which directly corresponds to a run of the system. It can be considered a counterexample for the invariant property.

The predicates are constructed in conjunctive normal form (CNF) from a digital circuit by applying Tseitin transformation [15]. It works by introducing auxiliary variables for internal signals. Then only local operations are necessary to get a satisfiability equivalent CNF: If G is an AND gate with inputs a, b and output c, then we construct the formula $c \leftrightarrow (a \wedge b)$, which is equivalent to $(\neg c \vee a) \wedge (\neg c \vee b) \wedge (c \vee \neg a \vee \neg b)$. A CNF for the whole circuit is given by the conjunction over the equivalences for each gate. Since this CNF is satisfied for an assignment iff it assigns consistent values to the signals in the circuit, we have to add a unit clause which sets the value of the output signal to true.

The advantage of applying Tseitin transformation is that the size of the predicates is linear in the size of the circuit. In contrast to that, the size, e. g., of OBDD representations, which are often used for symbolic model checking, might explode during the model checking process.

3 Bounded Model Checking for DTMCs

In this section we will turn our attention to how bounded model checking can be applied to discrete-time Markov chains in order to compute counterexamples for violated safety properties.

The differences to traditional bounded model checking are the following: (1) In general, it is not sufficient to compute a single path to a state which violates the invariant. Even the probability of the most probable path may be too small to exceed the bound p. Instead, we need a potentially large set of paths whose probability measure exceeds the given bound. (2) We normally cannot start from a circuit description of the system. Stochastic systems are usually modelled using a process algebraic description language from which the state space and the transition probabilities can be computed using techniques like parallel composition. An example for this formalism is the input language of the stochastic model checker PRISM [16], which we will later use for our experiments. Among other tools, PRISM is able to generate symbolic representations of the Markov chains. In particular, the transition probabilities are represented using Multi-terminal binary decision diagrams (MTBDDs) [17].

Since the matrix is normally sparse and well-structured, the symbolic representation using MTBDDs is in many cases much more compact than explicit representations. For this reason, we start with an MTBDD $\mathcal{P}(s,t)$ for the transition probability matrix, an OBDD $\mathcal{I}(s)$ for the initial state and an OBDD $\mathcal{L}_a(s)$ for each atomic proposition $a \in AP$ such that $\mathcal{L}_a(s) = 1$ iff $a \in L(s)$.

In the following, we will first describe some preprocessing steps, before we continue with the formula generation from OBDDs and the bounded model checking itself.

3.1 Preprocessing

Before we start the bounded model checking process, we reduce the computation of paths satisfying $a \mathsf{U} b$ to computing arbitrary paths ending in a b-state. Since we

do not need the probabilities for the path computation, we can use the associated Kripke structure for this operation. The OBDD-representation[1] $\mathcal{T}(s,t)$ of its transition relation can be obtained from the transition probability matrix $\mathcal{P}(s,t)$ by a threshold operation, i. e., $\mathcal{T}(s,t) = 1$ iff $\mathcal{P}(s,t) > 0$.

Removing Edges. We have to ensure that we will never find a path of which a proper prefix satisfies the path formula $a\mathsf{U}b$. This can be guaranteed if the b-states do not have any out-going edges. Furthermore we remove all out-going edges from states in which neither a nor b hold. This makes sure that all paths ending in a b state satisfy $a\mathsf{U}b$. Therefore we will call the b-states the *target states* of the system.

Often we can cut even more edges of the resulting graph: Only those a- (and target) states are relevant which can be reached from the initial state and from which a target state can be reached. These states can be determined using well-known symbolic graph traversal algorithms.

While the first edge cutting is mandatory for the correctness of the stochastic BMC algorithm, the latter is only an optimization. It may be skipped if the time or memory consumption of the reachability analysis is too high or if the resulting OBDD of the transition relation is considerably larger than the original one.

3.2 CNF Generation

From the OBDD representations $\mathcal{I}(s)$, $\mathcal{T}(s,t)$, and $\mathcal{L}_b(s)$, we have to generate the predicates in conjunctive normal form (CNF). They will be denoted by $I(s)$, $T(s,t)$, and $L_b(s)$. Since OBDDs can be considered as a special form of circuits such that each node corresponds to a multiplexer, we can apply Tseitin transformation to obtain a CNF as already described in Section 2.2. The transformation results in (at most) four clauses per OBDD-node. Our experiments have shown that it is beneficial to reduce the size of the CNFs as much as efficiently possible. We apply the following ideas:

- To avoid unnecessary auxiliary variables when both successor nodes are leaves. This situation results in equivalences of the form $(n_1 \leftrightarrow x)$, where n_1 is the auxiliary variable associated with an internal signal and x the label of an OBDD node. Then every occurrence of n_1 can be replaced by x.
- To use reordering (e. g., sifting [18]) to reduce the size of the OBDD for the transition relation.

3.3 Bounded Model Checking

Recall that for a given unrolling depth k the formula describing paths starting in the initial state s_I and ending in a target state is given by (cf. Eq. (1)):

$$BMC(k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge L_b(s_k).$$

[1] In the following, we use calligraphic letters $\mathcal{L}, \mathcal{P}, \mathcal{T}, \ldots$ for OBDDs and MTBDDs.

We feed it into a SAT-solver and try to find a satisfying solution. If the formula is unsatisfiable, there is no path of length k from the initial state to a target state. Otherwise, the satisfying assignment we get from the SAT-solver corresponds to a path in the original Markov chain which satisfies $a \mathsf{U} b$.

In contrast to model checking for digital or hybrid systems, we cannot stop once we have found a satisfying solution. Instead we have to continue until we have found enough paths such that their probability measure exceeds the given bound p.

For a boolean variable a, let a^1 denote the positive literal a and a^0 the negative literal $\neg a$. Let $s_{i,0}, \ldots, s_{i,n}$ be the variables of the state visited at time step i, and $v_{i,0}, \ldots, v_{i,n}$ their values assigned by the SAT-solver. To prevent the solver from finding the same solution again, we add the following clause to its clause database:

$$\left(\bigvee_{i=1}^{k} \bigvee_{j=0}^{n} s_{i,j}^{1-v_{i,j}} \right) \tag{2}$$

Thereby, we can ignore the auxiliary variables introduced by the Tseitin transformation, because their values are implied by the values of the original state variables. Furthermore we may start with step $i = 1$ instead of 0. Since the CNF contains unit literals which enforce that all solutions start in the initial state, the literals in the exclusion clause (2) which correspond to step 0 would be false and can therefore be left out completely. The same applies to step k if there is a unique target state.

We iterate the solution process until we have either found enough paths or the formula becomes unsatisfiable. If the latter is the case before we have found enough paths, we increase the unrolling depth k by one and continue.

Theorem 1. *If $s_I \not\models \mathcal{P}_{\leq p}(a \mathsf{U} b)$, the algorithm described above terminates and returns a set of paths whose probability measure is greater than p.*

Proof. (1) All paths which are found by the SAT-solver satisfy $a \mathsf{U} b$: Since all states which satisfy neither a nor b do not have any out-going edges anymore, the target state cannot be reached from such a state.

(2) The algorithm terminates after a finite number of steps: If $\mathcal{P}_{\leq p}(a \mathsf{U} b)$ is violated, there is a finite counterexample C (see [10]). This is found after at most $k = \max\{|\pi| \mid \pi \in C\}$ iterations. Since there is only a finite number of paths with length $\leq k$, the SAT-solver is only called a finite number of times.

(3) The probabilities are computed correctly: In our counterexample, no path is a prefix of another path. Hence the total probability is the sum of the probabilities of the single paths. □

3.4 Optimizing the BMC Process

Having all the paths computed by a SAT-solver is a time-consuming task, in particular when the counterexample consists of a large number of paths. Therefore we need to reduce the number of calls to the SAT-solver.

Minimal Distance Initial State → Target State. We would like to avoid unnecessary calls to the SAT-solver of which we can know in advance that they return UNSAT. This is the case if the current unrolling depth k is smaller than the length of the shortest path from the initial state to a target state. We therefore compute the minimal distance from s_I to a target state using the OBDD representation.

Algorithm 3.1: MinDist(Initial state $\mathcal{I}(s)$, Target states $\mathcal{L}_b(s)$)
Begin
$\mathcal{R}(s) \leftarrow \mathcal{I}(s)$, $old(s) \leftarrow \emptyset$, $dist \leftarrow 0$ (1)
While $\mathcal{R}(s) \neq old(s)$ **And** $\mathcal{L}_b(s) \wedge \mathcal{R}(s) = \emptyset$ **Do** (2)
$old(s) \leftarrow \mathcal{R}(s)$ (3)
$\mathcal{R}(s) \leftarrow \mathcal{R}(s) \vee \text{renameVariables}\big(t \rightarrow s \text{ in } \exists s : \mathcal{R}(s) \wedge \mathcal{T}(s,t)\big)$ (4)
$dist \leftarrow dist + 1$ (5)
End While (6)
If $\mathcal{L}_b(s) \wedge \mathcal{R}(s) \neq \emptyset$ **Then Return** $dist$ (7)
Else Return ∞ (8)
End

Later we will start with the minimal distance as the initial unrolling depth of the bounded model checking process.

Incomplete Assignments. Modern SAT-solvers detect satisfiability when all variables have been assigned a boolean value without resulting in a conflict. But often all clauses are satisfied before the solver has assigned each variable a boolean value.

If we are able to obtain a partial satisfying assignment in which n state variables are unassigned, this assignment corresponds to 2^n different paths. Furthermore, to exclude all these paths from the solution space only a single clause is necessary, which is shorter than a clause only excluding a single path.

Loop Detection. Let us assume we have found a path on which a state occurs twice, e. g., $\pi = s_I \xrightarrow{p_1} s_1 \xrightarrow{p_2} s_2 \xrightarrow{p_3} s_3 \xrightarrow{p_4} s_2 \xrightarrow{p_5} s_4$; s_I is the initial state, s_4 a target state. We can easily detect that there is a loop $s_2 \xrightarrow{p_3} s_3 \xrightarrow{p_4} s_2$. This loop can be taken arbitrarily often. Instead of returning a set of "flat" paths—one path for each unrolling of the loop—we construct counterexamples which consist of acyclic paths whose states may be annotated with loops (see Fig. 1).

Definition 6. *Let L be a set of loops, i. e., of paths σ such that $\sigma(0) = \sigma(|\sigma|)$ and for all $0 < i < |\sigma|$: $\sigma(i) \neq \sigma(0)$. An* annotated path $\hat{\pi}$ *is a pair $\hat{\pi} = (\pi, l)$ such that π is an acyclic path and $l : \{0, \ldots, |\pi|\} \rightarrow 2^L$ assigns each position on the path a set of loops with $\forall i \in \{0, \ldots, |\pi|\}\ \forall \sigma \in l(i) : \pi(i) = \sigma(0) = \sigma(|\sigma|)$.*

Such an annotated path $\hat{\pi}$ represents an infinite set of "flat" paths. For instance, the probability measure of the annotated path in Fig. 1 is

$$\Pr_{s_0}(\hat{\pi}) = \sum_{i=0}^{\infty} p_1 \cdot p_2 \cdot (p_3 \cdot p_4)^i \cdot p_5 = p_1 \cdot p_2 \cdot \frac{1}{1 - (p_3 \cdot p_4)} \cdot p_5.$$

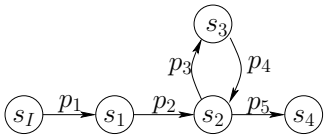


Fig. 1. A Path that is annotated with a loop

We have to take into account that this probability includes the probability of the path with 0 unrollings of the loop. This acyclic path has been found by the SAT-solver in one of the previous iterations.

If we have attached m different loops with probabilities $p_1, p_2, \ldots, p_m$ to the same state, we can increase the probability of the underlying acyclic path by the following factor:

$$\begin{aligned} p &= 1 + (p_1 + p_2 + \cdots p_m) + (p_1^2 + p_1p_2 + p_2p_1 + p_2^2 + p_1p_3 + \cdots + p_m^2) + \cdots \\ &= \sum_{n=0}^{\infty} \sum_{\substack{i_1,i_2,\ldots,i_m \in \mathbb{N} \\ i_1+i_2+\cdots+i_m=n}} \binom{n}{i_1\ i_2\ \cdots\ i_m} p_1^{i_1} p_2^{i_2} \cdots p_m^{i_m} \\ &= \sum_{n=0}^{\infty} (p_1 + p_2 + \cdots + p_m)^n = \frac{1}{1 - (p_1 + p_2 + \cdots + p_m)}. \end{aligned} \tag{3}$$

The advantages of generating sets of annotated paths are: (1) The user learns more about the structure of the system. This makes the diagnosis of faults easier. (2) The size of the counterexample can become considerably smaller since an infinite number of paths is represented by one annotated path. (3) The number of calls to the SAT-solver is reduced and thereby also the runtime of the bounded model checking process.

Path exclusion. Before calling the SAT-solver, we have to exclude all those paths from the solution space of the SAT-problem which originate from unrolling annotated paths we have already found in previous iterations and whose length is identical to the current unrolling depth k.

Assume, we have an annotated path $\hat{\pi} = (\pi, l)$, which is annotated with loops $\pi_1, \ldots, \pi_p$. We have to decide how often we have to unroll which loop in order to obtain a path of total length k. These unrollings correspond to the solutions of the following constraint with $r_1, \ldots, r_p \in \mathbb{N}$:

$$|\pi| + |\pi_1| \cdot r_1 + |\pi_2| \cdot r_2 + \cdots + |\pi_p| \cdot r_p = k. \tag{4}$$

We solve this constraint system using a simple enumeration algorithm. More sophisticated methods are certainly available.

If there are several loops attached to the same state, we need to exclude them in all possible orders. This can result in a large number of long clauses which need to be added to the SAT-solver's clause database. All paths generated by a single solution have the literals corresponding to the acyclic base path in common. The

remaining literals can be grouped according to the state of the base path their loop corresponds to. Let Π be the set of paths corresponding to a single solution of Eq. 4. They all have the acyclic base path $s_0 s_1 \dots s_{n-1}$ in common, and the states s_i occur at the same positions for all $\pi \in \Pi$. Each clause which excludes a path $\pi \in \Pi$ can therefore be split into C^{base}—the literals to exclude the base path—and $C_\pi^{s_i}$ for $i = 0, \dots, n-2$—the sets of literals to exclude the unrolling of the loops attached to s_i. Using the naive approach, the clauses to exclude the paths in Π are given by $\{C^{\text{base}}\} \times \{C_\pi^{s_0} \mid \pi \in \Pi\} \times \dots \times \{C_\pi^{s_{n-2}} \mid \pi \in \Pi\}$.

By introducing auxiliary variables, this set can be reduced using the following observation: Assume we have two clauses $(C \vee S_1) \wedge (C \vee S_2)$, whereby C, S_1, and S_2 are disjunctions of literals and C are the literals which both clauses have in common. We introduce a new auxiliary variable a and replace the two clauses by $(C \vee a) \wedge (\neg a \vee S_1) \wedge (\neg a \vee S_2)$. Applied to our path exclusion problem this leads to the following set of clauses:

$$\left\{C^{\text{base}} \dot\cup \{a_{s_0}, \dots, a_{s_{n-2}}\}\right\} \dot\cup \bigcup_{i=0}^{n-2} \left\{C_\pi^{s_i} \dot\cup \{\neg a_{s_i}\} \mid \pi \in \Pi\right\}. \tag{5}$$

The variables a_{s_i} are the auxiliary variables used to split off the loop unrolling at state s_i of the base path. The following lemma counts the number of clauses and literals for both approaches:

Lemma 1. *Let L be a set of loops; $\hat\pi = (\pi, l)$, an annotated path over L; $\pi = s_0 s_1 \dots, s_{n-1}$; and k, the current unrolling depth. Furthermore let $u : L \to \mathbb{N}$ be a solution of Eq. (4). Let $u(L') = \sum_{l' \in L'} u(l')$ for $L' \subseteq L$ and $l(s_i) = \{l_1^i, \dots, l_{n_i}^i\}$ the loops attached to state s_i. The number of clauses of the naive path exclusion vs. the optimized path exclusion is given by*

$$\#_c^{naive} = \prod_{i=0}^{n-2} \binom{u(l(s_i))}{u(l_1^i) \cdots u(l_{n_i}^i)} \qquad \text{vs.} \qquad \#_c^{opt} \le 1 + \sum_{i=0}^{n-2} \binom{u(l(s_i))}{u(l_1^i) \cdots u(l_{n_i}^i)}.$$

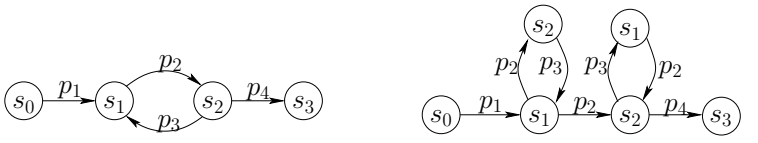
If bps denotes the number of bits per state, the total number of literals for both approaches is

$$\#_l^{naive} = \#_c^{naive} \cdot (k+1) \cdot bps$$

$$\#_l^{opt} \le (|\pi| + 1) \cdot bps + |\pi| + \sum_{i=0}^{n-2} \left(\binom{u(l(s_i))}{u(l_1^i) \cdots u(l_{n_i}^i)} \left(1 + bps \cdot \sum_{l' \in l(s_i)} |l'| \cdot u(l')\right)\right).$$

Correctness issues. We have to guarantee that we do not count the same path twice when we compute the probability measure of the set of annotated paths. This happens if the same path can be generated by unrolling loops in different ways.

Example 1. Let us assume the SAT-solver returns the path $s_0 \xrightarrow{p_1} s_1 \xrightarrow{p_2} s_2 \xrightarrow{p_3} s_1 \xrightarrow{p_2} s_2 \xrightarrow{p_4} s_3$. In principle, we could detect two loops: $s_1 \to s_2 \to s_1$ and $s_2 \to$



(a) A path with only one loop ...

(b) ... but we might detect two loops if we are not careful enough

Fig. 2. A Path with an embedded loop

$s_1 \rightarrow s_2$, which would result in the annotated path depicted in Fig. 2(b). One might think the probability of this annotated path was $p_1 \cdot \frac{1}{1-(p_2 \cdot p_3)} \cdot p_2 \cdot \frac{1}{1-(p_3 \cdot p_2)} \cdot p_4$, but this is not correct. The reason is that the same paths are generated by unrolling the first loop as by the second loop. The correct probability is therefore only $p_1 \cdot \frac{1}{1-(p_2 \cdot p_3)} \cdot p_2 \cdot p_4$. □

We can avoid to over-estimate the probability measure with the following two observations:

(1) On each path returned by the SAT-solver, there can be at most one loop which we may attach to a state of the underlying acyclic path. If it was a path with two loops, the solver would have returned two shorter paths—each with only one of the loops—in previous iterations. But then, we would have excluded the path with both loops in advance. If the path contains more than one state twice, this means that either the loop itself consists of sub-loops or that we have the situation depicted in Fig. 2(a). Both problems can be solved if we attach the loop to the first state which occurs twice on the path.

(2) We only attach loops to that position on the current path where the loop has been found by the SAT-solver. If the SAT-solver returns a path with a loop we can conclude that this path cannot be generated by unrolling any existing annotated path in our collection. So we may attach the loop to the corresponding basic path without over-estimating the correct probability measure.

The pseudocode of the optimized algorithm with loop detection is sketched in Algorithm 3.2. Starting with the minimal distance from the initial state to a target state, we iterate lines 1–14 until either the counterexample exceeds the probability bound p or the maximal unrolling depth has been finished. First, we generate the CNF for the current unrolling depth k (line 2), then we exclude all paths of length k, which originate from unrolling loops (line 3).

As long as the resulting formula is satisfiable, we exclude the newly found path from the solution space of the CNF. Furthermore we test if the path that corresponds to the satisfying assignment contains a cycle. If this is the case, split the path into the acyclic base path and the loop which we attach to the base path that is already contained in the counterexample (lines 8–10). Otherwise we insert the (acyclic) path into the counterexample (line 11). The algorithm terminates as soon as the probability measure of C is larger than p.

Algorithm 3.2: StochBMC(CNF I, CNF T, CNF L_b, Probability p)
Begin
For $k \leftarrow$ minDist **To** maxDist **Do** (1)
$\phi \leftarrow \text{CreateCNF}(I, T, L_b, k)$ (2)
$\phi \leftarrow \phi \wedge \text{ExcludePrecomputedPaths}(C, k)$ (3)
While ϕ is satisfiable **Do** (4)
$\pi \leftarrow \text{Solution2Path}(\phi)$ (5)
$\phi \leftarrow \phi \wedge \text{ExcludePath}(\pi)$ (6)
If π contains a cycle **Then** (7)
$(\pi_{\text{base}}, \pi_{\text{loop}}) \leftarrow \text{split}(\pi)$ (8)
$b \leftarrow \text{findBasePath}(C, \pi_{\text{base}})$ (9)
$C \leftarrow \big(C \setminus \{b\}\big) \dot{\cup} \big\{\text{attachLoop}(b, \pi_{\text{loop}})\big\}$ (10)
Else $C \leftarrow C \cup \{\pi\}$ (11)
If $\Pr_{s_I}(C) > p$ **Then Return** C (12)
End While (13)
End For (14)
Return "I could not generate a counterexample!" (15)
End

4 Experiments

We have implemented the BMC algorithm in C++ with the optimizations that were presented above. We use Cudd [19] for the OBDDs and Minisat [20] as state-of-the-art SAT-solver. We ran all our experiments on a Dual Core AMD

Table 1. Experimental results for the leader election protocol

			without loop detection				*with loop detection*				
N	K	bound	paths	depth	SAT-calls	time [s]	paths	loops	depth	SAT-calls	time [s]
3	2	0.99	66	16	79	0.15	6	12	8	23	0.05
3	3	0.99	143	12	152	0.39	24	66	8	95	0.20
3	4	0.99	276	8	281	0.72	60	202	8	267	0.51
3	5	0.99	589	8	594	3.08	120	451	8	576	2.41
3	6	0.99	1040	8	1045	3.97	210	808	8	1023	3.76
3	8	0.99	1979	8	1984	8.31	504	1455	8	1964	5.42
3	10	0.99	990	4	991	5.63	990	0	4	991	5.16
3	12	0.99	1711	4	1712	8.82	1711	0	4	1712	8.20
4	2	0.99	— TL —				8	64	10	78	0.23
4	3	0.99	— TL —				60	1215	10	1281	9.22
4	4	0.90	3903	12	3909	79.52	216	3223	10	3445	61.84
4	5	0.90	2123	10	2129	99.01	560	1483	10	2049	85.45
4	6	0.90	1167	5	1168	30.80	1167	0	5	1168	28.13
4	8	0.90	3687	5	3688	113.95	3687	0	5	3688	102.58
5	2	0.90	— TL —				10	204	12	221	1.06
5	3	0.90	9585	12	9592	382.00	180	7508	12	7695	272.06
5	4	0.90	23019	12	23026	2948.00	900	20270	12	21177	2438.50
5	5	0.90	2813	6	2814	1102.67	2813	0	6	2814	1044.63
6	2	0.90	— TL —				12	606	14	626	6.56
7	2	0.90	— TL —				14	1571	16	1594	31.46
8	2	0.90	— TL —				16	3796	18	3822	137.46

Table 2. Experimental results for the contract signing protocol

N	L	states	prob.	bound	SAT-calls	paths	depth	time [s]
5	3	53041	0.515625	0.50	513	512	31	49.67
5	4	73211	”	”	513	512	41	158.53
5	5	93381	”	”	513	512	51	278.90
5	6	115710	”	”	513	512	61	441.64
6	2	159742	0.5078125	0.50	2049	2048	25	127.12
6	3	258046	”	”	2049	2048	37	293.14
6	4	356350	”	”	2049	2048	49	758.91
6	5	454654	”	”	2049	2048	61	1778.75
7	2	737278	0.50390625	0.50	8193	8192	29	668.51
7	3	1196030	”	”	8193	8192	43	2231.64

Opteron$^{\text{TM}}$ processor with 2.4 GHz and 4 GB of main memory. For all experiments we set a time limit of 2 hours and a memory limit of 1 GB. The results of the experiments are listed in Table 1. Experiments which were aborted due to the time limit are marked with “— TL —”. We used the probabilistic model checker PRISM [16] to generate symbolic representations for benchmarks, all of which are available from the PRISM website `http://prismmodelchecker.org`.

The first benchmark we used is the synchronous leader election protocol by Itai and Rodeh [21]. It models a ring of N processors with a common clock. The goal is to elect a leader, i. e. a uniquely designated processor, by sending messages around the ring. The protocol proceeds in rounds. During each round the processors draw a random number from the range $\{1, \ldots, K\}$ as an id. If there is a unique maximal id, the processor with this id becomes the leader. Otherwise a new round is initiated. The Markov chains consist of a few hundred (for $N = 3$) up to 12 500 states (for $N = 5$, $K = 5$). We checked the property that finally a leader is elected. Since this happens with probability 1.0 when starting in the initial state, we generated paths to provide a certificate that the given bounds (0.99 and 0.90, respectively) are indeed exceeded.

One can see that the version with loop detection always performs better than the version without. In those cases where the counterexample does not contain any loops, both versions of the algorithm perform similarly. For other instances loop detection turns out to be essential for counterexample generation. E. g., for the `leader`-benchmark with $N = 4$ and $K = 2$, the algorithm without loop detection found 239 138 paths (maximal length: 29) with a total probability measure of 0.980 077 2 before the time limit was exceeded. In contrast to this, the optimized algorithm only needed 8 paths with 64 loops and an unrolling depth of 10 to reach the probability bound of 0.99.

In addition to the leader election protocol, we applied BMC to a contract signing protocol [22,23]. Its purpose is to exchange pieces of information (e. g. digital signatures) between two parties A and B over a network. One important property that such a protocol should exhibit is fairness. This means, whenever B has obtained A's commitment, B cannot prevent A from getting B's commitment. For the variant of the protocol we used, the actual probability to reach an unfair state is slightly larger than 0.5. The violated property for which we

generated counterexamples is $\mathcal{P}_{<\frac{1}{2}}(\text{true}\,\mathsf{U}\,\text{unfair})$. Since the optimized algorithm did not find any loops before reaching the probability bound and since the results for both variants do not differ substantially, we only show the results of the algorithm without loop detection in Table 2. N denotes the number of data pairs to exchange; and L, the size of each piece of data.

The results for the contract signing protocol show that we are able to handle quite large Markov chains with more than 10^6 states. More states are possible if we decrease the probability bound for the counterexample.

5 Conclusion

In the previous sections we have shown how bounded model checking can successfully be applied to generate counterexamples when invariant properties of Markov chains are violated. By returning not simply "flat" paths, but paths which are annotated with loops, we make not only the algorithm more efficient, but also the counterexamples more useful for the designer: (1) In many cases fewer paths are needed such that the given probability bound is exceeded and (2) the structured counterexamples provide more information about the system: Each annotated path represents an acyclic execution trace together with possible deviations from this direct path in the form of loops. Additionally, bounded model checking returns a counterexample which consist of shortest possible paths (although they might be not the most probable ones).

Points for further research are to make the detection of loops more powerful such that loops themselves may consist of loops. This will reduce the size of the counterexample further. Other optimizations aim at reducing the number of SAT-calls, e. g. by re-using loops on different annotated paths. Furthermore, optimization techniques known from conventional bounded model checking should be transferred to the stochastic world to speed-up the solution of the SAT-instances. The extension of bounded model checking to Markov decision processes (MDPs) will be another topic of our future research.

References

1. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
2. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
3. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: 11th IEEE Int'l Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp. 207–212. IEEE CS, Los Alamitos (2008)
4. Hojati, R., Brayton, R.K., Kurshan, R.P.: BDD-based debugging of design using language containment and fair CTL. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 41–58. Springer, Heidelberg (1993)
5. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: 32nd Design Automation Conference, pp. 427–432 (1995)

6. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Symposium on Logic in Computer Science (LICS), pp. 19–29. IEEE CS, Los Alamitos (2002)
7. Gurfinkel, A., Chechik, M.: Proof-like counter-examples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 160–175. Springer, Heidelberg (2003)
8. Aljazzar, H., Hermanns, H., Leue, S.: Counterexamples for timed probabilistic reachability. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 177–195. Springer, Heidelberg (2005)
9. Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 33–51. Springer, Heidelberg (2006)
10. Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)
11. Damman, B., Han, T., Katoen, J.P.: Regular expressions for PCTL counterexamples. In: Rubino, G. (ed.) 5th QEST, Saint-Malo, France, pp. 179–188. IEEE CS, Los Alamitos (2008)
12. Andrés, M.E., D'Argenio, P., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Haifa Verification Conference. LNCS. Springer, Heidelberg (2008)
13. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
14. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
15. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic, Part 2, 115–125 (1970)
16. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
17. Fujita, M., McGeer, P.C., Yang, J.C.Y.: Multiterminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design 10(2/3), 149–169 (1997)
18. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: IEEE/ACM Int'l Conf. on Computer Aided Design (ICCAD), pp. 42–47 (1993)
19. Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.4.1. University of Colorado at Boulder (2005)
20. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
21. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation 88(1), 60–87 (1990)
22. Even, S., Goldreich, O., Lempel, A.: A randomized protocol for signing contracts. Communications of the ACM 28(6), 637–647 (1985)
23. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. Journal of Computer Security 14(6), 561–589 (2006)

Author Index